分类号：TP31　　单位代码：10335

学　　号：11821067

浙江大学

博士学位论文

中文论文题目：面向 Linux 外设的虚拟化关键技术研究

英文论文题目：Research on Key Technologies of Virtualization for Linux-based Peripherals

申请人姓名：刘强

指导教师：周亚金研究员

合作导师：

学科（专业）：网络空间安全

研究方向：系统安全，软件安全

所在学院：计算机科学与技术学院

论文递交日期　2023 年 9 月 8 日

致谢

人们喜欢做选择，但是常常不知道如何选择。如果让我获得一种超能力，我想预知未来。如今博士毕业在即，已知晓五年前的全部未来，回头望，也很难讲清楚过去五年中的每一个选择是否恰当。既已选择，不再遗憾。回顾过去的一切选择，不仅仅是自己的判断，同行者也有很大的影响。在此，我感谢他们跟我一起塑造了今天的我。

罗森林，潘丽敏，刘晓双，魏源，王鹏伟，王迎圆，沈晓琳，陈焰，何博远，朱添田，杨润青，许同，冷雪，杜学超，金羚，熊春霖，李振源，谢子怡，徐立恒，朱梦凡，阮琳琦，吴尚，程帅，陈安东，宋怡焕，卜凯，周亚金，周侠，李嘉奇，王丁玎，吴斯韦，陈元，郭景怡，吴华茂，姜木慧，张岑，刘杨，马麟，徐金焱，陈卓，胡宇峰，李鹏飞，卜誉杰，向阳曦，贺博文，程静，管一，王克，王达豹，王刚，张文龙，吴磊，纪守领，王琴应，申文博，杨昱天，常瑞，张卓若，席少柯，周金梦，李晓晨，丁楚颖，何冠局，单法鹏，杨雨润，Mathias Payer，Marcel Busch，Flavio Toffalini，Gwangmu Lee，Adrian Herrera，Atri Bhattacharyya，Ahmad Hazimeh，Nicolas Badoux，Luca Di Bartolomeo，Florian Hofhammer，Andres Sanchez，吕涛，丰至瑶，张驰斌，Natascha Fontana，许多，彭辉，姜植元，Manuel Egele，郑晗，Prashast Srivastava，Hossein Moghaddas，Majid Salehi，翁谋毅，胡锦鑫，段林瑞，朱奕宸，于清晨，张培贤，杜媛媛，张晨晨，陈剑辉，高逸铭，吴家奇，吴杨，刘悦，陈梓彤，杜云涛，戴逸展，杨震，张路波，谭寅，于虎，童晓梦，孙立力，孙羊羊，吴江玮，陈燕钦，Phillip Mao，Zurab Tsinadze，徐坚浩，Alexander Bulekov，Jeremy Lai，HyungSoek Han，梁振凯，吕绘林，张语珊，刘天天，胡小蕙，杨明俊，宁晨，王浩宇，张锋巍（以上排名不分先后）。

摘要

物联网在日常生产生活中应用广泛。其中，基于 Linux 内核的物联网设备（简称 Linux 物联网设备）占比大、安全风险高，其安全性亟待分析和加强。由于 Linux 物联网设备获取困难、可扩展性差、可调式性差，需采用虚拟化技术将这些设备的固件重新托管在虚拟执行环境中。Linux 物联网设备的虚拟化有两个核心目标。首先，保证虚拟执行环境的保真性，使虚拟 Linux 物联网设备与真实设备接近；其次，维护虚拟执行环境的安全性，防止虚拟 Linux 物联网设备之间互相影响。因为 Linux 物联网设备主要由众多且复杂的外围设备（简称 Linux 外设）组成，所以虚拟 Linux 外设也是虚拟执行环境的主要组成部分和最大的攻击面。因此，本文为实现前述两个核心目标紧紧围绕 Linux 外设展开了研究，分别提出了 1）基于模型引导内核执行的新技术，通过构建高保真的虚拟 Linux 外设保证整个虚拟执行环境的保真性；2）基于依赖感知消息模型的新技术，通过模糊测试虚拟 Linux 外设维护整个虚拟执行环境的安全性。

首先，本文提出了一种基于模型引导内核执行的新技术。该技术通过分析外设相应的内核子系统建立状态机、分析相应的底层驱动获取状态转移条件，半自动化地构建 Linux 物联网设备的内核的虚拟执行环境，解决了已有虚拟执行环境构建技术无法处理该内核导致保真性丧失的问题。实验表明，原型系统 FirmGuide 生成了 9 个要求完整功能的 I-型虚拟外设和 64 个要求最小功能的 II-型虚拟外设；模拟了 26 个片上系统；重新托管了超过 95% 的 Linux 物联网设备的内核，涵盖了两个架构和 22 个内核版本。

其次，本文提出了一种基于依赖感知消息模型的新技术，实现了高效的虚拟外设模糊测试框架。该框架将虚拟外设的输入依赖建模为消息内依赖和消息间依赖，通过从虚拟外设源代码中提取消息内依赖、利用精心设计的突变器学习消息间依赖，解决了输入依赖带来的测试效率低下的问题。实验表明，原型系统 ViDeZZo 涵盖了两个虚拟机管理程序、四个架构、五个设备类别和 28 个虚拟外设，具有可扩展性；与之前的工作相比，更快地达到了可比的代码覆盖率，具有高效性；成功地复现了 24 个已报道的安全缺陷，发现了 28 个新的安全缺陷。我们提供的 7 个补丁已被合并在虚拟外设代码仓中。

最后，通过上述两种创新方法，本文最终实现了高保真和高安全的 Linux 物联网设备虚拟执行环境，支持了 Linux 物联网设备漏洞分析和漏洞挖掘（模糊测试）等应用，

促进了 Linux 物联网设备安全研究的进一步发展和实际应用。

关键词：Linux 物联网设备；虚拟执行环境；虚拟外设；保真性；安全性

Abstract

The Internet of Things (IoT) is widely used in our daily life. Among them, Linux-based IoT devices are the most prevalent and of high security risks, and thus their security needs to be analyzed and strengthened urgently. Since hardware is not always available, not scalable, and hard to debug, virtualization technology is required to rehost Linux-based IoT devices on virtual execution environment (VEE). Virtualization for Linux-based IoT devices has two objectives. First, keep the fidelity of the VEE, that is, the VEE should be as close as possible to the physical Linux-based IoT device; second, keep the security of the VEE, that is, each virtual Linux-based IoT device should be well isolated. Since Linux-based IoT devices consist of multiple complicated peripherals, i.e., Linux-based peripherals, virtual Linux-based peripherals become the main component and the biggest attack surface of the VEE. Therefore, to realize the two objectives, focusing on the Linux-based peripherals, we propose two new technologies, respectively, 1) model-guided kernel execution, which ensures the fidelity of the whole VEE by constructing high-fidelity virtual Linux-based peripherals; 2) dependency-aware message model, which maintains the security of the whole VEE by fuzzing virtual Linux-based peripherals.

First, we propose a new technique named model-guided kernel execution to keep the fidelity of the VEE. This technique builds state machines with Linux subsystems and extracts state transition conditions from the corresponding low-level drivers, addressing the problem that existing technologies cannot rehost Linux-based IoT device kernels. Evaluations show that the prototype FirmGuide generates 9 fully functional Type-I virtual peripherals and 64 minimally functional Type-II virtual peripherals, supporting 26 System on Chips; successfully rehosts over 95% of the Linux-based IoT device kernels, covering two architectures and 22 kernel versions.

Second, we propose a new technique named dependency-aware message model and design a virtual peripheral fuzzing framework to keep the security of the VEE. We model the input dependencies to the virtual peripherals as intra-message and inter-message dependencies. The framework extracts the intra-message dependencies from the virtual peripheral source code and learns the inter-message dependencies by three new mutators, addressing the problem that

existing virtual peripheral fuzzers are low efficient due to the input dependencies. Evaluations show that the prototype ViDeZZo is both scalable, covering two hypervisors, four architectures, five device classes, and 28 virtual devices and efficient, achieving competitive code coverage faster compared to previous work. ViDeZZo reproduced 24 reported bugs and discovered 28 new bugs. We also provided seven patches merged into the hypervisor mainstream.

Through the above two novel methods, this paper finally realizes a high-fidelity and high-security VEE to analyze and mine vulnerabilities for Linux-based IoT devices, which helps with the further development and application of Linux IoT device security research.

Keywords: Linux-based IoT devices; Virtual execution environment; Virtual peripherals; Fidelity; Security

目录

图目录

表目录

1 绪论

1.1 研究背景和意义

物联网（Internet of Things）描述了由传感器、计算设备、软件（常称为固件）等通过互联网或者其他通信网络互相连接并交换数据的一种系统，在运输和物流、工业制造、健康医疗、智能环境（家庭、办公、工厂）、个人和社会领域等应用广泛[1]。其中，基于 Linux 内核的物联网设备（简称 Linux 物联网设备），如路由器、IP 摄像头和打印机等，约占总数的 71%，又因使用弱密码和未打补丁软件等，安全风险高，已成为攻击者的主要目标之一[2-3]。由于 Linux 物联网设备获取困难、可扩展性差、可调式性差，已有的分析进而加强 Linux 物联网设备安全性的动态程序分析需采用虚拟化技术，将 Linux 物联网设备重新托管（rehosting）在以虚拟机管理程序（hypervisor）为基础的虚拟执行环境中。Linux 物联网设备虚拟化具有两个核心目标。首先，保持虚拟执行环境的保真性（fidelity），使其与实际的 Linux 物联网设备尽可能接近；其次，维护虚拟执行环境的安全性（security），防止虚拟 Linux 物联网设备突破其虚拟执行环境进而影响其他设备。一个高保真性的、高安全性的 Linux 物联网设备虚拟执行环境将有助于打破已有技术的瓶颈，促进相关安全研究的开展和应用，如模糊测试、漏洞分析、自动化蜜罐部署等。

1.2 研究内容和创新点

Linux 物联网设备虚拟化具有两个核心目标，即保持虚拟执行环境的保真性和维护虚拟执行环境的安全性。要实现这两个核心目标，Linux 物联网设备的主要组成部分 — 外围设备（简称 Linux 外设）至关重要。因为 Linux 外设众多且复杂，实现高保真性的虚拟执行环境需通过专门的设计构建高保真的虚拟 Linux 外设，即从虚拟执行环境的构建出发从设计上保证高保真性。因为众多且复杂的虚拟 Linux 外设又是虚拟执行环境最大的攻击面，实现高安全性的虚拟执行环境需要通过高效的测试得到高安全的虚拟 Linux 外设，即从虚拟执行环境的测试出发在开发阶段保证高安全性。因此，本文紧紧围绕 Linux 外设开展研究分别提出了，1）基于模型引导内核执行的新技术，通过构建高保真的虚拟 Linux 外设保证整个虚拟执行环境的保真性；2）基于依赖感知消息模型的新技

术，通过模糊测试虚拟 Linux 外设加强整个虚拟执行环境的安全性。

1.2.1 基于模型引导内核执行的虚拟执行环境构建研究

Linux 物联网设备的固件由 Linux 内核和根文件系统组成。但是，本研究开展时，已有的虚拟执行环境构建方法无法重新托管 Linux 物联网设备固件中的原生内核，明显地导致保真性缺失，因此再也无法发现原生内核里的安全问题[4-5]。如何重新托管 Linux 物联网设备固件中的原生内核是一个亟待解决的问题。本文提出了一种叫做模型引导内核执行的新技术，以重新托管 Linux 物联网设备的原生内核。原型系统 FirmGuide 已经开源[6]，在这个基础上，模糊测试、漏洞分析已成功地被应用在 Linux 物联网设备上。

本研究将成功地重新托管 Linux 物联网设备的原生内核定义为成功地把该原生 Linux 内核启动并进入用户态。而成功地把该原生 Linux 内核启动到用户态的关键就是处理众多且复杂的 Linux 外设。因此，构建高保真的虚拟执行环境的就是要回答如何构建高保真的虚拟外设。对此，本研究基于以下三个观察提出了基于模型引导内核执行的虚拟外设构建技术。1. 重新托管 Linux 内核只需要少数高保真的 Linux 内核启动过程要求功能完整的虚拟外设（称为 I-型虚拟外设），其他不要求完整功能的虚拟外设（称为 II-型虚拟外设）只需正确的初始化值来实现它们的最小功能。2. Linux 内核对不同类型的外设都有明确的抽象（状态机），如针对中断控制器的 Linux 内核的中断子系统。3. 将从 Linux 内核的子系统中提取的状态机、从相应的底层驱动提取的状态转移条件组合起来，即可实现高保真的 I-型虚拟外设。

总的来说，该技术通过分析 Linux 内核的源代码，半自动地建立 I-型虚拟外设。具体来说，该外设模型由两部分组成：模型模板和模型参数。模型模板是能够与 Linux 内核交互的状态机，是根据 Linux 内核对该类型外围设备的抽象层即相应的子系统手动建立的。状态机定义了所有的状态和状态转换表，但把状态转移条件留成空白。状态转移条件是利用 Linux 底层驱动代码中的使用符号执行提取出的模型参数生成的。在提取模型参数时，II-型虚拟外设所需要的初始值可以一并自动化地分析出来。

上述部分称为“离线模型生成”，进一步，在生成所需的 I-型和 II-型虚拟外设之后，要想重新托管一个 Linux 物联网设备，还需要根据它的硬件列表将各个虚拟外设组合起来生成一个完整的虚拟执行环境，此部分称为“在线内核启动”。

实验结果表明，FirmGuide 生成了 9 个虚拟 I-型外设和 64 个虚拟 II-型外设，支持

了 26 个片上系统。用从互联网上下载的 Linux 物联网设备固件进行的实验表明，它成功地重新托管了超过 95% 的 Linux 内核，涵盖了两个架构和 22 个内核版本。经测试，FirmGuide 成功地支持了模糊测试、漏洞分析等应用。

该研究的创新点如下： 1. 总结了重新托管 Linux 物联网设备内核的最小条件，区分了 I-型和 II-型外围设备。2. 提出了一种名为模型引导内核执行的新技术对 I-型外围设备建模。3. 原型系统 FirmGuide 支持了在 Linux 物联网设备上的模糊测试、漏洞分析。

1.2.2 基于依赖感知消息模型的虚拟执行环境模糊测试研究

Linux 物联网设备的虚拟执行环境既要提供高保真性，也要隔离虚拟 Linux 物联网设备以免互相影响。例如，运行在虚拟 Linux 物联网设备中恶意软件或者攻击者不能突破其自身虚拟执行环境而影响其他虚拟 Linux 物联网设备。虚拟外设是虚拟执行环境的重要组成部分，它数目众多且功能复杂，是虚拟执行环境最大的攻击面。如果能提前发现虚拟外设中的漏洞并修复这些漏洞，那么虚拟执行环境的安全性将获得保障。模糊测试是挖掘漏洞最有效的方法，但是本研究开展时，已有的虚拟外设模糊测试方法，受限于消息内依赖和消息间依赖，效率不高。如何高效地模糊测试虚拟外设是一个亟待解决的问题。本文针对开源的虚拟机管理程序 QEMU 和 VirtualBox 展开了研究，提出了一个新的依赖感知消息模型的虚拟外设模糊测试框架 ViDeZZo[7]。该原型系统已经开源，成功地复现了 24 个已报道的安全缺陷，发现了 28 个新的安全缺陷。

总的来说，该依赖感知消息模型解决了虚拟外设模糊测试时的两个挑战。1. 消息内依赖：虚拟外设消息内部通常包含多个字段，这些字段可能是相互依赖的。2. 消息间依赖：虚拟外设消息之间有先后顺序，存在依赖关系。具体来说，该框架为了解决消息内依赖性设计了一个新颖的、轻量级的描述性语法，半自动地注释消息内依赖；为了解决消息间依赖性设计了三类新的突变器，自动地学习消息间依赖。

与之前的工作相比，ViDeZZo 既具有可扩展性（涵盖了两个虚拟机管理程序、四个架构、五个设备类别和 28 个虚拟外设），又具有高效性（更快地达到了可比的代码覆盖率）。成功地复现了 24 个已报道的安全缺陷，发现了 28 个新的安全缺陷，获得了一个 CVE，这些安全缺陷涵盖了多样的类型。此外，由我们提供的 7 个补丁已经被接收。

该工作的创新点如下： 1. 总结了虚拟外设模糊测试的难点，区分了消息内依赖和消息间依赖。2. 提出了一种依赖感知消息模型的模糊测试框架，同时考虑了消息内和消

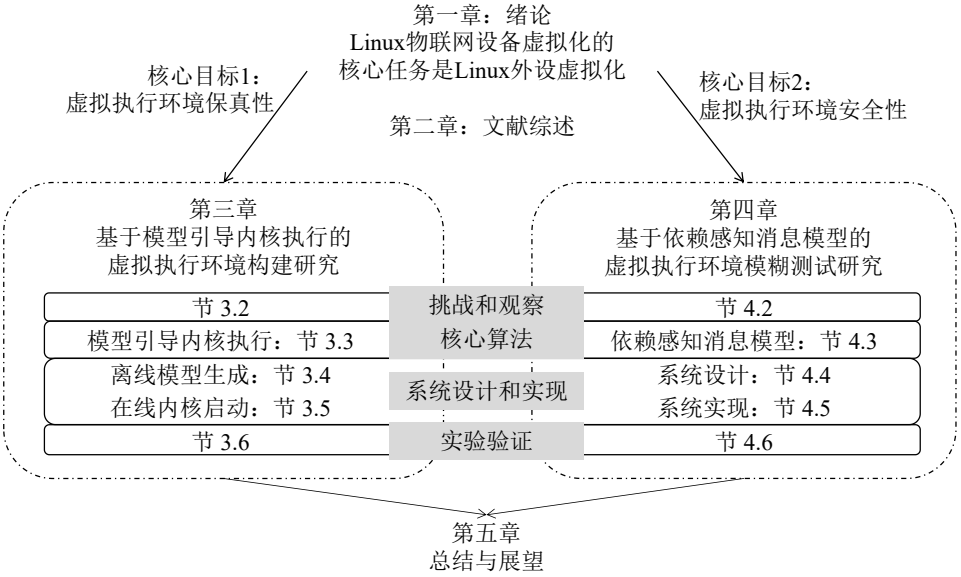


图 1.1 论文总结架构示意图

息间依赖。3. 开发了原型系统 ViDeZZo，高效地和大规模地测试了虚拟外设。

综上所述，本文通过基于模型引导内核执行的虚拟执行环境构建技术、基于依赖感知消息模型的虚拟执行环境测试技术，提供了高保真、高安全的虚拟执行环境，成功地支持了 Linux 物联网设备的模糊测试和漏洞分析。

1.3 本文的组织结构

本文的总体架构如图 1.1所示。

第一章是本文的绪论，介绍了本文的研究背景和意义，以及本文的研究内容和创新点。本章首先介绍了 Linux 物联网设备虚拟化的必要性；然后解释了 Linux 物联网设备虚拟化的两个目标：保真性（fidelity）和安全性（security）；最后介绍了本文为了实现 Linux 物联网设备虚拟化的两个目标，紧紧围绕 Linux 物联网设备中的众多且复杂的外围设备，在虚拟执行环境构建和虚拟执行环境测试两方面的研究内容和创新点：通过模型引导内核执行构建高保真的虚拟外设来保证虚拟执行环境的高保真性、通过依赖感知的模糊测试框架测试虚拟外设来维护虚拟执行环境的高安全性。

第二章针对本文提出的两方面的研究内容即“虚拟执行环境构建”和“虚拟执行环境测试”开展了文献综述，介绍了相关技术的背景知识，探讨了国内外研究人员在这两个领域上的研究现状以及本文所提出技术的在该研究发展过程中的位置。

第三章介绍了基于模型引导内核执行的新技术，回答了如何构建高保真的 Linux 物联网设备内核的虚拟执行环境的问题。本章首先介绍了重新托管 Linux 物联网设备内核所需的背景知识，解决该问题所面临的挑战和支持所提出新技术的观察；然后，介绍了模型引导内核执行算法的示例和基本概念；紧接着，介绍了该算法中的两个主要部分的设计和实现，离线模型生成和在线内核启动。实验结果表明，该技术能成功地重新托管 Linux 物联网设备，并支持对 Linux 物联网设备的模糊测试、漏洞分析。

第四章介绍了基于依赖感知消息模型的模糊测试框架，回答了如何构建高安全的虚拟执行环境的问题。本章首先介绍了摆脱虚拟设备输入依赖所需要的背景知识，解决该问题所面临的挑战和支持所提出新技术的观察；然后，介绍了基于依赖感知消息模型的具体内容；紧接着，介绍了该模糊测试框架的系统设计和系统实现。实验结果表明，应用该技术可以大规模地和高效地对虚拟设备开展模糊测试。

第五章总结了本文的研究工作，展望了未来的研究方向。

2 文献综述

2.1 引言

Linux 物联网设备构建于集成了 CPU、内存和外围设备的片上系统之上，如路由器、IP 摄像头和打印机等，运行由 Linux 内核和根文件系统组成的固件，提供着丰富的功能。2016 至 2017 年爆发的 Mirai 僵尸网络警示人们关注这些 Linux 物联网设备的安全性。Mirai 僵尸网络利用这些 Linux 物联网设备使用弱密码和未打补丁软件等特点，不幸地感染了 100 多万台 Linux 物联网设备[8]。分析 Linux 物联网设备的安全性有两条技术路线。其一是静态程序分析。静态程序分析通过扫描该 Linux 物联网设备的固件可以检测出弱密码和未打补丁软件的使用，但是要进一步地挖掘和分析该固件中的漏洞，甚至利用蜜罐技术捕获最新的攻击行为，静态分析就显得捉襟见肘。其二就是动态程序分析。动态程序分析如模糊测试和动态调试等可以高效地挖掘和分析该固件中的漏洞，能促进 Linux 物联网设备的安全性分析的进一步发展，解决了静态程序分析的局限性。虚拟化技术则进一步解决了 Linux 物联网设备获取困难、可扩展性差、可调式性差等问题，促进了 Linux 物联网设备的动态分析的进一步发展。

Linux 物联网设备虚拟化具有两个核心目标，即保持虚拟执行环境的保真性和维护虚拟执行环境的安全性。为了更好地理解这两个核心目标，本章将从虚拟执行环境构建和虚拟执行环境测试两方面出发介绍相关的背景知识、国内外研究人员在这两个领域上的研究现状，以及本文所提出的技术在该研究发展过程中的位置。

2.2 虚拟执行环境构建国内外研究现状

物联网设备虚拟执行环境构建的目的是在虚拟执行环境中运行该设备的固件，也称之为重新托管。重新托管可以分为两步：第一，构建单独的虚拟外设；第二，将单独的虚拟外设组合成一个完整的虚拟执行环境。

第一，重新托管 Linux 物联网设备的核心任务是构建虚拟外设，尤其是关键的外围设备如中断控制器和定时器。Linux 物联网设备构建于集成了 CPU、内存和外围设备的片上系统之上，从数量上看，外围设备是片上系统的主要组成部分。在这些外围设备

表 2.1 重新托管技术发展脉络表

	软硬件交互（保留硬件）	软硬件交互（摆脱硬件）	直接建模	硬件抽象层	符号执行
2014	Avatar[9] Prospect[10]				
2015	Surrogates[11]				
2016					
2017					
2018					
2019		Pretender[12]	P2IM[13]	HALucinator[14]	
2020					Laelaps[15]
2021		Conware[16]	DICE[17] **FirmGuide**	para-rehosting[18] ECMO[19]	μEMU[20] Jetset[21]
2022		Jun et. al.[22]	Zhou et. al.[23]	MetaEmu[24]	Fuzzware[25] MetaEmu[24]
2023					ICICLE[26] Ember-IO[27] DevFuzz[28]

```
compatible = "plxtech,nas7820";
cpu@0;                          // processor
memory;                         // memory
ic@47001000 {                   // peripheral 1
    compatible="arm,arm11mp-gic";
    reg = <0x47001000 0x1000>; // MMIO memory space <start, size>
};
ethernet@41000000;              // peripheral 2
```

图 2.1 OX820 NAS7820 片上系统的设备树的示例

中，中断控制器和定时器是两个最重要的外围设备。外围设备使用中断来通知处理器正在发生的事情，附属于处理器的中断控制器负责传递中断。在中断控制器通知处理器一个中断被触发后，处理器再从它那里获取中断请求号（Interrupt Request Number，简称 IRQn），并跳转到相应的中断服务例程（Interrupt Service Routine，简称 ISR）。一个片上系统通常需要两种定时器。一个连接到中断控制器，用于周期性地产生中断。另一个从不产生中断，Linux 内核通过定期读取这个定时器的计数寄存器来维护时间。中断控制器和定时器使能了 Linux 内核的进程调度、中断和异常处理、时间管理等基础功能。

第二，利用设备树中硬件列表可以将单独的虚拟外设组合成一个完整的虚拟执行环境。Linux 物联网设备运行着由 Linux 内核和根文件系统组成的固件，但除此之外，Linux 物联网设备的固件往往还使用设备树来描述一个片上系统的硬件信息。Linux 内核在启动之初使用它来正确地初始化外围设备。设备树使用名为 compatible 的属性来记录一个硬件的型号。如图 2.1所示，根 compatible 是片上系统的型号，次级是外围设备（即中

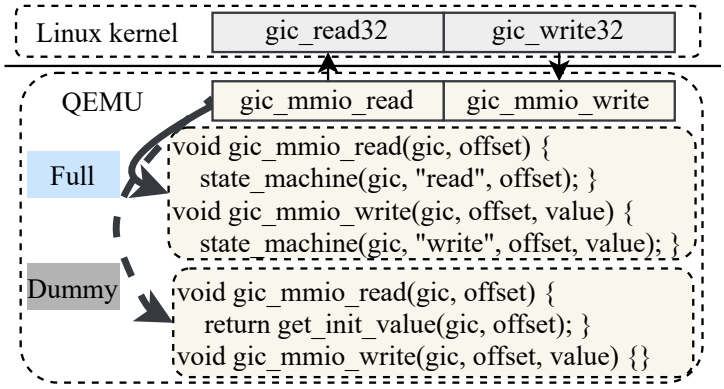


图 2.2 QEMU 外围设备建模示例：全功能和无功能的外围设备模型

断控制器）的型号。此外，设备树记录了一个外围设备的丰富的信息，包括其 MMIO 范围、IRQn、输入时钟频率等。一旦单个虚拟设备构建完毕，即可依据设备树组织各个外围设备的方式将不同的虚拟设备组合成为一个虚拟执行环境。

以 QEMU 为例，在图 2.2中，单个虚拟设备为其寄存器实现了读和写的回调函数。当 Linux 物联网设备的内核从一个 MMIO 地址读时，读回调函数将被调用。写回调函数的工作原理与此类似。一个全功能的虚拟设备有一个模拟外围设备功能的状态机。没有这样的状态机就是无功能的虚拟外设。尽管如此，这类模型需要适当的值来初始化其 MMIO 地址空间。QEMU 将多个虚拟设备组合起来即能提供一个完整的虚拟执行环境。

梳理了重新托管的步骤之后，本节继续梳理了针对物联网设备重新托管的相关研究，如表 2.1所示。因虚拟设备组合基本是一个已解决的问题，不同的重新托管的方法即关注在如何发展一种通用的技术构建单个虚拟设备上。

早期，为了实现重新托管，Avatar[9]、Prospect[10]和 Surrogates[11]等在虚拟执行引擎和物理设备之间传递数据流和控制流，同时保留着硬件。尽管这些方法保真性较高，但受到硬件可得性和调试接口可用性的影响较大。

随着技术的发展，研究人员逐渐摆脱了对硬件的依赖，向着全系统重新托管迈进。Pretender[12] 通过对固件和物理设备之间的交互进行建模，设法取代真实设备。后来，P2IM[13]通过收集指令序列填充了预定义的模型。HALucinator[14]识别了固件中的硬件抽象层（Hardware Abstract Layer，简称 HAL），并在没有硬件仿真的情况下替换了 HAL 中的关键函数。P2IM 和 HALucinator 成功地在没有任何物理设备辅助下对物联网设备的固件进行了动态分析，同时也保持了较高的保真度。具体如下。

Pretender 记录了固件和其硬件之间的真实交互，并使用机器学习和模式识别技术每

个外围设备创建模型。具体来说，Pretender 先将交互记录按照外围设备的地址空间进行分组，然后推断何时触发一个中断和是哪个 MMIO 事件触发了这个中断，再判断 MMIO 寄存器符合“简单存储模型”、“模式模型”、“递增模型”、“只写模型”的哪一种，最后将已获得的信息组合起来作为一个虚拟执行环境。类似的，Conware[16]通过这些交互自动推断状态机，对外围设备进行建模。Jun 等还利用这些交互发现了“隐藏的内存映射”[22]。

P2IM 基于一个观察，只要虚拟机执行环境在固件需要时提供可接受的（不一定是真的）外设输入，固件就可以在没有真实的或完全模拟的外围设备下执行。模型推导过程包含两个步骤。第一，根据嵌入式设备的手册定义一个描述寄存器、中断和内存布局的抽象模型；第二，根据探索性固件执行自动推断出实例这个抽象模型所需要的信息，如一个存储器在内存中的具体位置、已经使能的中断等。进一步的，DICE 建立了 DMA 的模型并将其实例化[17]。后来，Zhou 等利用自然语言处理技术将记录在外围设备手册中人类语言翻译成一组结构化的条件行动规则，在固件运行时检查、执行和连锁这些规则，动态地合成一个外围设备模型[23]。

HALucinator 基于固件开发者经常使用 HAL 简化他们的工作这一事实，通过替换 HAL 功能将固件与硬件解耦。具体来说，该方法首先通过二进制分析在固件中找到库函数，然后根据外围设备的类型，开发处理函数，替换掉这些库函数。例如，网卡处理函数可以通过 DMA 发送一个以太网帧。类似想法的文章还有 para-rehosting[18]、MetaEmu[24]以及一个针对 VxWorks 的案例研究[29]。

与此同时，基于符号执行的方法也开枝散叶。μEMU[20]试图在单个外围设备寄存器的粒度模拟固件的执行。具体来说，它对固件进行符号执行，将未知的外围设备寄存器作为符号，求解出外围设备寄存器的可能的值，储存在一个知识库中，在动态固件分析过程中被参考。同期相似想法的文章还有 Laelaps[15]、Jetset[21]、Fuzzware[25]、MetaEmu[24]、ICICLE[26]、Ember-IO[27] 和 DevFuzz[28]。基于符号执行构建的虚拟设备的保真度最低。

Linux 物联网设备的固件由原生内核和用户应用程序两部分组成，上述技术不能直接运用于 Linux 物联网设备。实际上，在本论文所提技术之前，针对 Linux 物联网设备重新托管的研究局限在用户应用程序上。Firmadyne 使用一个自定义的 Linux 内核（非固件原生内核）提供了一个动态分析用户应用程序的工具[30]。其他类似的系统[31-35]也专注于用户应用程序。这些工作的共同点在于丢弃了 Linux 物联网设备中的 Linux 内核，削弱了虚拟执行环境的保真度，导致一些隐藏在其中的问题无法被发现[4-5]。

本文所提 FirmGuide（详见章 3）考虑了 Linux 物联网设备中的原生的 Linux 内核，并采用了一种新的方法：通过分析 Linux 内核子系统建立外围设备的状态机，通过分析外围设备驱动获得状态转移条件，对外围设备进行了精确的建模，填补了技术空白。后来，与 HALucinator 想法类似，ECMO[19]利用 LuaQEMU[36]监控并改写代码执行的能力，用已有的外围设备驱动替换了固件内核内原生的驱动，像体外人工肺一样接管了固件内核的执行，进一步使能了 Linux 物联网设备虚拟执行环境中的网络功能。

虚拟执行环境的保真性需要与应用有关。例如，模糊测试不需要较高保真性的虚拟执行环境，而漏洞分析就需要的较高的保真性。本文所提技术 FirmGuide 提供了更高保真性的虚拟执行环境，使能了 Linux 物联网设备的漏洞挖掘和漏洞分析。

2.3 虚拟执行环境测试国内外研究现状

虚拟机管理程序是虚拟机 (Virtual Machine，简称 VM) 和物理硬件之间的核心软件抽象层，既提供了虚拟执行环境，又被赋予了隔离虚拟机的关键安全责任。但是，虚拟机管理程序中的严重漏洞层出不穷[37-39]，其中，利用模糊测试发现的漏洞就有数百个。

模糊测试（fuzzing）是一种软件测试技术，旨在发现软件的缺陷，经过多年的发展，已经发现数以千计的漏洞[40-41]，是最有效的漏洞挖掘方法之一。典型的模糊测试技术是一种灰盒测试技术，它仅仅通过了解程序有限的信息来辅助测试。该技术由三大部分组成[42]：输入生成器（input genenator）、执行器（executor）和反馈机制（feedback）。模糊测试器的输入有两类生成方法，一是生成式的，基于语法或模板；一是变异式的，基于突变器（mutator）。执行器则需要将输入传递给目标程序，并运行目标程序。反馈通常有两种，第一种是代码覆盖率，如果执行完一个输入之后，全局代码覆盖率增加，那么就可以称这个输入是有趣的（interesting），会把这个输入加入语料库（corpus）；另一种反馈是漏洞检测器，如地址消毒剂（Address Sanitizer，简称 ASAN）[43]和内核地址消毒剂（Kernel Address Sanitizer，简称 KASAN）[44]。当前，针对用户应用程序的主流的模糊测试框架有 AFL++[45]、libFuzzer[46]和 honggfuzz[47]，针对操作系统的是 Syzkaller[48]。

因为虚拟设备是虚拟执行环境最大的攻击面，虚拟执行环境测试的核心任务是测试虚拟设备。将模糊测试应用到虚拟设备上，与用户态应用程序不同，需要处理虚拟设备消息。虚拟机管理程序拦截来自客户的 PIO 和 MMIO 操作并将其请求转发给虚拟外

```
void ehci_opreg_write(physaddr addr, uint64_t val, uint32_t size) {
  switch (addr) {
    case USBCMD:           // do something
    case PERIODICLISTBASE: // do something
    case ASYNCLISTADDR:    // do something
```

图 2.3 虚拟外设中 I/O 回调示例

表 2.2 虚拟外设消息和它们的描述汇总

Messages	Description
PIO ops	PIO read/write operations
MMIO ops	MMIO read/write operations
Memory ops	Memory allocation, read/write, or free
Clock ops	Time adjustments

设中预定义的回调。如图 2.3所示，对 USB EHCI 的寄存器的 MMIO 写操作被重定向到 QEMU 中的回调 ehci_opreg_write()，接下来，控制流会根据 addr 的值流向不同的专门处理程序。PIO 和 MMIO 操作遵循驱动虚拟外设的协议，称为虚拟外设消息。一个虚拟外设消息，例如，一个 MMIO 写操作，定义了客户与虚拟外设的通信方式。如表 2.2所示，除了 PIO 和 MMIO 消息，虚拟外设消息还包括与内存有关的操作，以操纵主内存空间，以及与时钟有关的消息，以调整当前时间[49]。

接下来本节梳理了针对虚拟机管理程序的模糊测试的相关研究，如图 2.4所示。

2017 年之前，学术界和工业界[50-52] 尝试了除了模糊测试之外的其他方法，并未获得较大的突破。2017 年，Deng 等人首先提出了初具模样的模糊测试器 VDF，通过记录内核启动时 MMIO 消息和变异这些消息来测试虚拟机管理程序中的虚拟外设。即使 VDF 只发现了一个新漏洞，它启发了研究人员提出更先进的虚拟机管理程序模糊测试器。

首先，虚拟机管理程序模糊器变得更快。2020 年，Schumilo 等人提出了 HyperCube，在客户机操作系统中构建了一个具有高吞吐量的输入解释器[53]。2021 年，Schumilo 等人扩展了他们的工作并发布了一个新的模糊测试器 Nyx，重用了这个解释器[54]。同年，Pan 等人提出了 V-Shuttle[55] 和 Bulekov 等人提出了 Morphuzz[56]，都摆脱了对客户机操作系统的依赖，而是在与虚拟机管理程序相同的进程中对虚拟外设进行模糊测试，由于减少了虚拟机管理程序和虚拟机管理程序之间的上下文切换，速度甚至更快。

其次，虚拟机管理程序模糊测试器变得更加高效。HyperCube 是一个黑盒模糊测试器。由于引入了覆盖反馈和手动编码语法，Nyx 更加高效[54]。覆盖率反馈已成为对虚拟机管理程序进行模糊测试的基本要求。同时，V-Shuttle 和 Morphuzz 都指出 DMA 访问

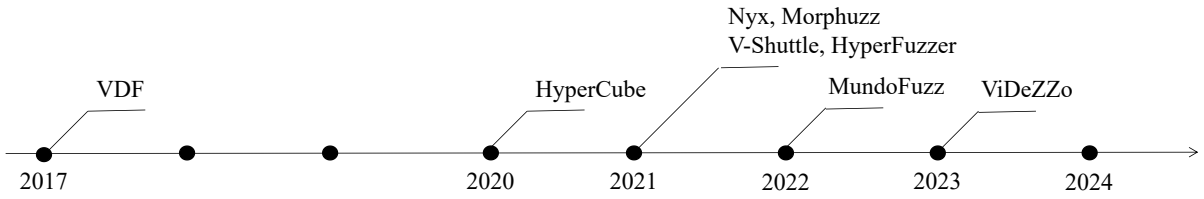


图 2.4 虚拟机管理程序模糊测试发展脉络图

表 2.3 输入依赖推断技术发展一览表

	语法描述	自动推断		
		从历史数据中自动推断	依据覆盖反馈自动推断	依据程序状态自动推断
2016	Syzlang			
2017	DiFuzz[58]			
2018		Moonshine[59]		
2019				
2020				HFL[60]
2021	SyzGen[61]		Morphuzz[49] VShuttle[55]	Healer[62]
2022	KSG[63]	MundoFuzz[57]		StateFuzz[64]
2023	SyzDescribe[65] NGFuzz[66]		**ViDeZZo**	Actor[67]

的内容对于虚拟外设测试至关重要，并提出了各自的启发式的方案。但是在当时，虚拟机管理程序的模糊测试仍然面临着效率不高的问题，这是因为已有的模糊测试器忽略了消息内依赖和消息间依赖。本文所提 ViDeZZo（详见章 4）分别提出了新的消息内注释，允许从虚拟机管理程序源代码中半自动提取消息内依赖性，和新的多级变异器，允许模糊器在模糊测试期间自动学习消息间依赖关系。2022 年，Myung 等人提出 MundoFuzz 采用了差分的思想提取 IO 序列中的依赖关系[57]，进一步补充了消息间依赖的推断方法。消息间依赖注释仍然有着提高的空间，不同于缓慢的自主学习和间接地从 IO 序列中推断，利用静态分析算法或许可以直接从虚拟机管理程序源代码中直接构建消息间依赖。

由于虚拟机管理程序模糊测试是一个较新的研究方向，如表 2.3所示，本文还梳理了包含操作系统在内的输入依赖推断的相关研究。这些研究主要沿着两个方向逐步推进：直接的语法描述和间接的自动推断。两种方法各有优缺，相互促进，协同发展。

Syzkaller 使用 Syzlang[68]语言描述系统调用的依赖关系。使用这个语言，可以较好地定义系统调用的参数和返回值，体现在两方面。一方面，如果一个参数是一个资源，如文件句柄，它可以描述是哪个系统调用创建了这个资源作为返回值；另一方面，如果一个参数一个结构体，它可以描述这个结构体的组成。起初，这些描述是人工开发的，

多篇文章探讨了自动生成系统调用描述的方法[58,61,63,65]和完全抛弃该描述的可行性[66]。本文首次提出了注释虚拟外设消息的语法，有效地支持了消息内依赖。

使用语法描述输入依赖的代表性工作具体如下。DiFuzz 静态分析内核驱动代码，提取 ioctl 接口涉及的命令和相关的数据结构。SyzDescribe 总结了 Linux 内核核心和驱动程序之间的编程惯例，涉及内核驱动如何被初始化以及它的相关接口如何被构建，根据这些知识生成所需的能静态地初始化内核驱动的系统调用描述。NGFuzz 利用了访问用户空间内存和管理文件描述符的 API，直接从模糊测试器（例如 libFuzzer）传递系统调用参数（包括数据和文件描述符）。与 Syzkaller 不同，不需要对系统调用进行复杂的描述，最终达到了可比的覆盖率还发现了新的漏洞。

除了语法描述之外，另一大类方法是自动推断。

一种自动推断的方法是从历史数据中提取信息。Moonshine[59]记录真实负载产生的系统调用并应用静态分析进行“蒸馏”，除去无用的信息但保留整个系统调用的依赖关系。MundoFuzz[57]也有类似的策略，利用记录的真实负载产生的虚拟外设消息进行差分测试判断依赖关系。具体来说，MundoFuzz 会把当前正确的虚拟外设消息序列调整成一份不正确的虚拟外设消息序列，将正确的与不正确的分别执行，检查两者的覆盖率。如果覆盖率不同，则“调整”的部分不能忽视，存在一定的依赖关系。

如果不使用历史数据，一种简单的方式是将对依赖的猜测交给模糊测试引擎[49,55]。本文在支持虚拟外设消息的基础上，首次提出了三组不同的突变器，利用模糊测试的遗传进化特性自动的学习消息间依赖。

直接从模糊测试器学习输入依赖是自然而然的，但是有时候效果不佳。HFL[60]、Healer[62]、StateFuzz[64]和 Actor[67]将输入与程序内部状态对应起来，通过分析程序内部状态的依赖，就能找到输入的依赖。探索程序内部状态，也就是探索程序的状态空间，有多种方式，如研究共享变量、状态变量和特定的行为。HFL 首先用静态分析提取不同的系统调用对同一个内存对象的写和读，再通过混合符号执行来确认这种依赖关系是存在；除了确定系统调用的顺序之外，这种方法还能把依赖关系拓展到系统调用的参数上。Healer 首先定义了何为“影响关系”：如果一个系统调用的执行能够通过修改内核的内部状态而影响到另一个系统调用的执行路径，则该前一个系统调用对另一个系统调用具有影响，又通过静态的和动态的方法，提取了这种关系。StateFuzz 通过静态分析识别一些表示内核状态的状态变量，将它们的运行时的值的变化作为状态转换的标志，作为

新的反馈，指导模糊测试。Actor 首先定义具有高层次语义的操作为“action”，如分配一个内核对象，并在模糊测试的过程中，识别出这样的“action”；再分析不同的 action 是否操作了同一个内核对象；最后根据专门的漏洞模板，将不同的“action”组合起来，直接生成一个有可能触发漏洞的测试用例。

输入依赖自动推断技术发展到现在，已经走入探索状态空间的深水区。如上述文章所示，状态空间是抽象的，针对不同的问题、不同的方法都有不同的表现形式。如何定义一个状态、如何推断状态转移条件、如何利用所得状态辅助模糊测试或者其他安全应用，已有了一些答案，但仍需继续探索。

2.4 本章小结

本章在虚拟执行环境构建和虚拟执行环境测试这两方面分别介绍了相关技术的背景知识、国内外研究人员在这两个领域上的研究现状，以及本文所提出的技术在该研究发展过程中的位置。一方面，物联网设备虚拟执行环境构建的相关研究已经开展多年，且日趋成熟。本文所提出的基于模型引导内核的技术填补了针对 Linux 物联网设备的内核的虚拟执行构建的技术空白，启发了更多的针对 Linux 物联网设备的内核的相关研究。另一方面，物联网设备虚拟执行环境测试的相关研究仍处于初始发展阶段。本文所提出的基于依赖感知消息模型的技术使虚拟设备模糊测试更加高效，同时引导了研究人员关注消息内和消息间依赖，以及更复杂的如基于状态的依赖推断。

3 基于模型引导内核执行的虚拟执行环境构建研究

Linux 物联网设备虚拟化的第一个核心目标是保证虚拟执行环境的保真性。因为 Linux 物联网设备主要由复杂的 Linux 外设组成，所以实现高保真性的虚拟执行环境需通过专门的设计构建高保真的虚拟 Linux 外设，从虚拟执行环境的构建出发从设计上保证高保真性。因此，本章提出了基于模型引导内核执行的新技术，通过构建高保真的虚拟 Linux 外设保证整个虚拟执行环境的保真性。

3.1 引言

近年来，随着物联网（或嵌入式）设备的普及，这些设备的固件（firmware）已经成为最常见的攻击目标之一[69-71]。它们中的许多都集成了 Linux 内核[30,72]，每年都有新的漏洞[73]。更糟糕的是，设备供应商没有及时将安全补丁从内核主线向后移植到固件中[74-76]，将有漏洞的设备暴露在野外。一旦被利用，这些设备可以被攻击者完全控制。

重新托管（Rehosting）也被称为仿真，用在模拟器内，如 QEMU，加载和运行分析目标，如固件的 Linux 内核，并提供监控目标运行状态的能力。之后，可以应用各种安全分析工具，例如，漏洞分析和模糊测试。在 QEMU 中运行桌面系统的 Linux 内核不是一个问题。然而，嵌入式设备往往使用不同类型的片上系统（System on a Chip，简称 SoC），而这些片上系统目前不被 QEMU 所支持。因此，在模拟环境中动态运行嵌入式 Linux 内核仍然是一个未解决的研究问题。

本章专注于重新托管嵌入式 Linux 内核，为它们的动态分析奠定基础。尽管研究人员在固件重新托管方面已经取得了一些进展，但这些工具不能满足我们的目的。首先，它们针对的是固件的用户应用程序，而不是 Linux 内核[30-31,33,77]。另外，Firmadyne[30]使用可在 QEMU 中加载的定制 Linux 内核，重新托管用户应用程序。由于加载的内核与真实的内核不同，它不能用来分析原始的 Linux 内核。其次，它们关注裸机系统的非 Linux 内核[13-14,18]，或者利用真实硬件[9]进行分析，针对 Linux 内核是不可扩展的。据我们所知，目前还没有系统可以大规模地重新托管嵌入式 Linux 内核。

重新托管嵌入式 Linux 内核是一项挑战。首先，启动过程依赖于多个外围设备。例

如，在分析了 Linux 内核中 1639 个设备树文件后，我们发现每个嵌入式设备平均有 32 个外围设备。其次，同一类型的外围设备通常有不同的硬件设计。Linux 内核支持的外围设备有数千种，逐一实现它们是不现实的。第三，外围设备接口具有语义多样性。一个外围设备通常向 Linux 内核暴露几个接口（硬件寄存器）来控制其内部状态，每个寄存器都有其特定的语义。一个成功的重新托管需要对这些接口的语义有所了解。这些挑战使得手动重新托管一个嵌入式 Linux 内核成为一项繁琐和容易出错的任务。

在本章中，我们旨在提高在 QEMU 中重新托管嵌入式 Linux 内核的能力。通过这样做，现有的基于 QEMU 的动态分析工具可以很容易地应用于分析嵌入式 Linux 内核。我们有三个观察来半自动地生成外围设备模型。

- 重新托管 Linux 内核只需要对少数外围设备（本章称为 I-型外围设备）进行完全仿真，例如，中断控制器和定时器是两个需要完全功能仿真的 I-型外围设备。对于其他外围设备（本章称为 II-型外围设备），我们只需要用正确的初始化值来仿真它们的最小功能，例如网卡和闪存。
- Linux 内核对不同类型的外围设备都有明确的抽象。例如，Linux 内核的中断子系统抽象了中断控制器的常见动作，并将这些动作定义为回调函数，供低级设备驱动程序注册。对于 Linux 内核的时间子系统和定时器外围设备也有类似的设计。
- 定义明确的抽象和内核外围设备交互一起可以描述外围设备接口。例如，中断子系统绘制了中断控制器的内部状态。如果我们可以使用内核外围设备交互来识别哪个回调函数被执行，那么我们就可以正确地驱动外围设备的内部状态。

随后，我们提出了一种新的技术，叫做模型引导的内核执行。它通过分析 Linux 内核的源代码，半自动地建立 QEMU 还不支持的 I-型外围设备模型。外围设备模型由两部分组成：一般模型模板和特定模型参数。模型模板是根据 Linux 内核对该类型外围设备的抽象层手动建立的，例如，中断子系统，这是一次性的工作。这些抽象层帮助我们构建能够与 Linux 内核交互的状态机。状态机定义了所有的状态和状态转换表，但把转换条件（事件）留成空白。然后我们从外围设备的驱动代码中提取模型参数来填补这些空白，以生成转换条件。这些参数是由符号执行自动生成的。II-型外围设备所需要的初始值在符号执行的同时生成。然后，外围设备模型可以用于合成一个 QEMU 虚拟机重新托管该嵌入式 Linux 内核。

我们开发了一个名为 FirmGuide 的原型系统，它有两个组成部分。一个是离线模型

生成，分析 Linux 内核的源代码，以半自动化地生成外围设备模型。另一个是在线内核启动，用设备树检查 Linux 内核的硬件依赖性，合成 QEMU 虚拟机重新托管 Linux 内核。请注意，FirmGuide 在离线模型生成中需要 Linux 内核的源代码来生成外围设备模型，但在在线内核启动中不需要源代码来重新托管固件内的 Linux 内核。为了让社区参与进来，我们发布了 FirmGuide 的源代码[6]。

为了评估有效性，我们生成了 9 个 I-型外围设备模型和 64 个 II-型外围设备模型（小节 3.6.2），并为每个嵌入式 Linux 内核合成了相应的 QEMU 虚拟机。通过虚拟机，我们把从互联网下载的嵌入式 Linux 内核重新托管[78-79]。一方面，5947 (96.11%) 的 Linux 内核被成功地重新托管（进入用户空间），而它们都不能在原生 QEMU 中重新托管（小节 3.6.3）。另一方面，重新托管的 Linux 内核覆盖了 26 个片上系统、两个架构和 22 个内核版本（小节 3.6.4）。请注意，重新托管的 Linux 内核版本不一定与生成外围设备模型所使用的内核版本相同。这个结果，加上重新托管的 Linux 内核的数量，说明了我们系统的可扩展性。此外，我们使用 Linux 测试项目（Linux Test Project，简称 LTP）[80] 来测试重新托管的 Linux 内核的功能，显示了部署安全应用程序的可行性（小节 3.6.5）。

我们进一步介绍了两个应用，以展示合成的 QEMU 虚拟机支持下的安全应用场景。首先，我们分析了 6 个 Linux 内核 CVEs。在 QEMU 调试能力的帮助下，我们分别成功地触发了 5 个和利用了它们的 4 个。这展示了使用合成的 QEMU 虚拟机来触发、理解和利用重新托管的嵌入式 Linux 内核的漏洞的能力。其次，我们移植了两个模糊测试工具 UnicoreFuzz[81]和 TriforceAFL[82]，以证明在重新托管的嵌入式 Linux 内核上支持其他动态分析工具（也就是模糊测试）的能力。这些应用本身并不是我们的贡献，但可以展示 FirmGuide 的使用场景。尽管如此，如果没有 FirmGuide 重新托管嵌入式 Linux 内核的能力，就很难将这些安全应用部署起来。

3.2 挑战和观察

我们的目标是提高在 QEMU 中重新托管嵌入式 Linux 内核的能力。为了重新托管这些嵌入式 Linux 内核，我们首先与一位经验丰富的嵌入式系统工程师讨论了他通常是如何支持一个新设备的。根据他的经验，我们惊讶地发现，基于 Linux 的固件的源代码，扩展它以支持一个新的物联网设备（直到内核成功启动，例如，弹出一个用户控制台窗

口）只需要调整某些外围设备的驱动代码。这一事实意味着，重新托管嵌入式 Linux 内核的任务可能并不像它看起来那么复杂（可能只需要几个关键的外围设备）。为了更好地理解这一点，我们系统地分析了 Linux 内核的启动过程和 QEMU 内一个外围设备的模拟机制。我们把发现的关键挑战和对重新托管的观察总结为以下几点。

重新托管嵌入式 Linux 内核的三个挑战。

- 一个真实的嵌入式设备通常有多个外围设备，手工模拟这些外围设备是很繁琐和容易出错的。对 OpenWRT 固件的 1639 个设备树的统计显示，每个嵌入式设备的外围设备数量从 5 到 76 不等，平均在 32 个左右。
- 嵌入式 Linux 内核在不同的片上系统之间共享常见的外围设备类型，例如中断控制器或定时器，但这些外围设备遵循不同片上系统供应商的不同硬件规范，手动实现所有外围设备是不现实的。
- 每个外围设备的接口（硬件寄存器）都有不同的语义，因此使外围设备模型的构建更具挑战性。这种多样性体现在两个方面。一方面，通过外围设备接口，Linux 内核可以检测到外围设备的状态并控制其行为。外围设备保持其内部状态并根据不同的接口做出反应。另一方面，一个外围设备接口本身也有语义。例如，中断控制器将每个中断源的状态（已屏蔽、未屏蔽等）保持在一个寄存器内。这个寄存器的一个位的翻转可以导致硬件内部状态的改变。建立一个成功的外围设备模型通常需要了解其内部状态、状态转换和硬件寄存器语义。

重新托管嵌入式 Linux 内核的三个观察。

- 实现我们的目标只需要对少数几个外围设备进行全功能仿真。当 CPU 进入用户模式执行 run_init_process() 时，我们定义嵌入式 Linux 内核成功被重新托管。尽管一个成功的重新托管过程涉及几十个不同的外围设备（平均 32 个），但它们可以分为两种类型。具体来说，I-型外围设备包括那些与 Linux 内核有复杂交互的外围设备，需要模拟其全部功能。这种类型的外围设备包括中断控制器、定时器和通用异步收发传输器（Universal Asynchronous Receiver/Transmitter–UART）。一个流行的 UART，也就是 NS16550A，在 QEMU 中得到了很好的支持，可以直接被重用。然而，其余的 I-型外围设备，即中断控制器和定时器，需要全功能的仿真。II-型外围设备只需要最低限度的模拟（一个具有适当初始值的 MMIO 内存区域）。它们只需要在 Linux 内核启动过程中读取特定的外围设备寄存器（MMIO 读）时提供

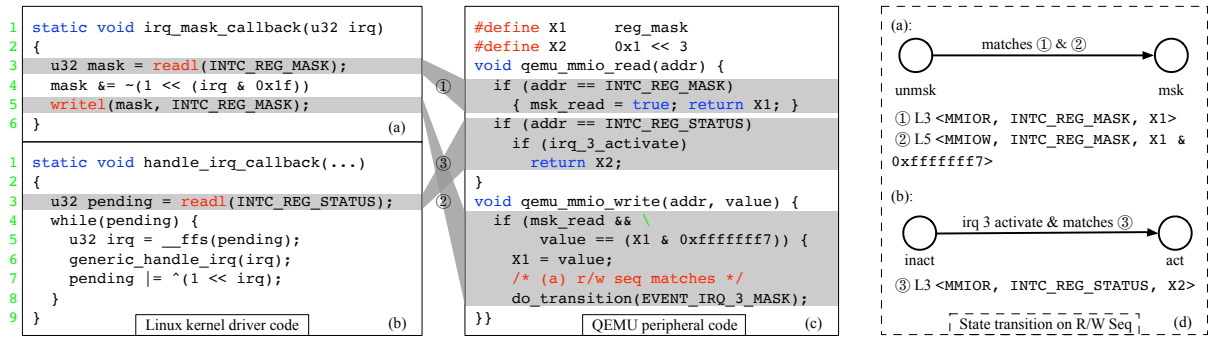


图 3.1 使用 R/W Seq 指导内核执行的例子：（a），（b）分别显示了识别和控制，（c）显示了外围设备模型中的 R/W Seq 匹配，（d）详细介绍了 R/W Seq 的状态转换；(a)，(b) 中的 MMIO 操作用灰色标记，宏 INTC_REG_XXX 代表 MMIO 寄存器地址，___ffs 是 “查找第一个匹配项” 的缩写

合适的值。II-型外围设备的内部状态不需要被模拟，例如网卡、闪存等。

- Linux 内核对不同类型的外围设备有明确定义的抽象模型。Linux 内核核心（上层）将一种类型的外围设备抽象为具有一套通用动作的设备。每个设备驱动（下层）都必须实现这些动作。对于 I-型外围设备（中断控制器和定时器），上层的抽象层分别是中断和时间子系统。下层的实现层是外围设备的低级设备驱动程序。例如，在中断子系统中，每个中断控制器是一个 struct irq_domain 实例，常见的动作是回调函数，如 irq_domain->irq_mask()、irq_domain->irq_unmask()、irq_domain->irq_ack() 等。一个低级别的设备驱动程序提供了这些功能的实现。在本章的其余部分，我们使用关键函数来表示这些回调函数。

- 定义良好的抽象模型和内核外围设备交互可以结合起来描述外围设备接口。上述的内核抽象启发我们在 QEMU 中建立一个外围设备模型，结合内核与外围设备的互动来指导内核的执行，使其成功地重新托管（因此被称为模型指导下的内核执行）。具体来说，外围设备模型是一个从上层（子系统）总结出来的状态机。它定义了状态和可用的状态转换。这个状态机可以从 Linux 内核源代码中手动提取，是一次性的工作。此外，外围设备模型需要识别执行的关键函数，然后作出相应的反应。一个关键的直觉是，从 Linux 内核到外围设备的 MMIO 读/写序列（Read/Write Sequence，简称 R/W Seq）是代表一个关键函数执行路径的签名。它们可以用于识别 Linux 内核执行的关键函数的路径，并通过在 MMIO 读请求中返回特定的值来控制 Linux 内核的后续执行。R/W Seq 可以通过符号执行自动推断出来。

3.3 核心算法之模型引导的内核执行

3.3.1 模型引导内核执行的实例

图 3.1展示了一个使用 R/W Seq 来指导内核执行的例子。MMIO 操作被标记为 <MMIOR/MMIOW, addr, expr>，其中 MMIOR/MMIOW 是读/写操作，addr 是读/写地址，expr 是 Linux 内核从外围设备模型读/写的值。为了简单起见，我们只考虑 IRQn 为 3。

图 3.1(a) 显示了一个简化的 irq_mask() 回调函数，以演示如何从外围设备的角度识别其执行。Linux 内核调用它来屏蔽一个特定的中断源（参数为 irq）。给定一个具体的 irq 值，例如 3，R/W Seq 是 <MMIOR, INTC_REG_MASK, X1>，<MMIOW, INTC_REG_MASK, F(X1)>，其中 X1 是 MMIO 的读值，$F(X1) = X1\&\texttt{0xfffff7}$。如图 3.1(c) 所示，通过匹配 R/W Seq，外围设备认识到 Linux 内核正在调用 irq_mask()。进一步地，外围设备可以通过检查 X1（返回给 Linux 内核）和 value（写给外围设备）是否满足方程式 $F(X1)$ 来弄清楚 IRQn 是否为 3。在与 R/W Seq 相匹配后，外围设备识别出事件 “Linux 内核屏蔽了编号为 3 的中断源”，并相应地转移其状态。

图 3.1(b) 是对 handle_irq() 回调的实现，以演示如何从外围设备的角度控制其执行。一旦 Linux 内核收到来自中断控制器的中断请求，它就会调用 handle_irq() 来调度从中断控制器获得的 IRQn 的请求。R/W Seq 是 <MMIOR, INTC_REG_STATUS, X2>，其中 X2 代表 MMIO 的读值。外围设备可以通过提供一个精心设计的 X2 来控制内核执行。在图 3.1(c) 中，在匹配 R/W Seq 后，外围设备返回 0x1 « 3 并告诉 Linux 内核，在读取 INTC_REG_STATUS 后，“中断源 3 应该被触发”。

最后，如图 3.1(d) 所示，我们发现 R/W Seq 可以作为外围设备模型中的状态转换条件，引导内核的执行走向一个特定的代码路径。直观地说，我们可以通过手动建立状态和转换的一般外围设备模型（使用 Linux 内核的抽象）并提取 R/W Seq 作为转换条件（使用符号执行来分析关键函数）来创建外围设备的完整状态机。

3.3.2 模型引导内核执行的概述

基于上述观察，我们提出了一种新的方法，称为模型引导的内核执行来仿真 I-型外围设备。我们把外围设备模型分成两部分：模型模板（独立于外围设备）和模型参数（依赖于外围设备）。具体来说，对于每一种类型的外围设备，我们从 Linux 内核子系

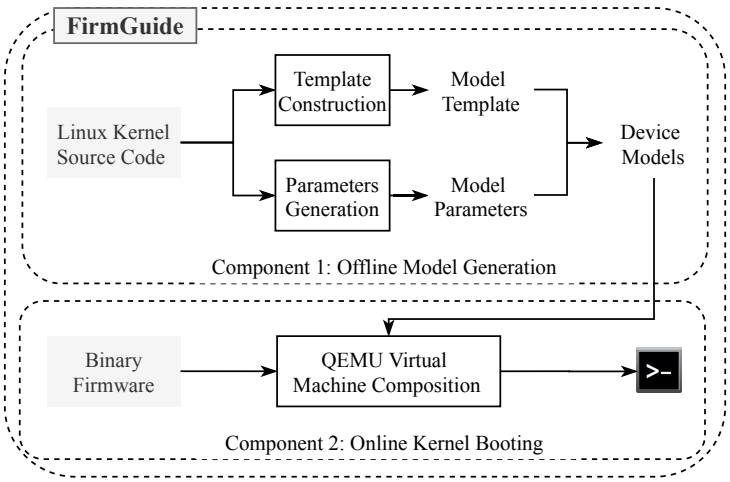


图 3.2 FirmGuide 的架构示意图

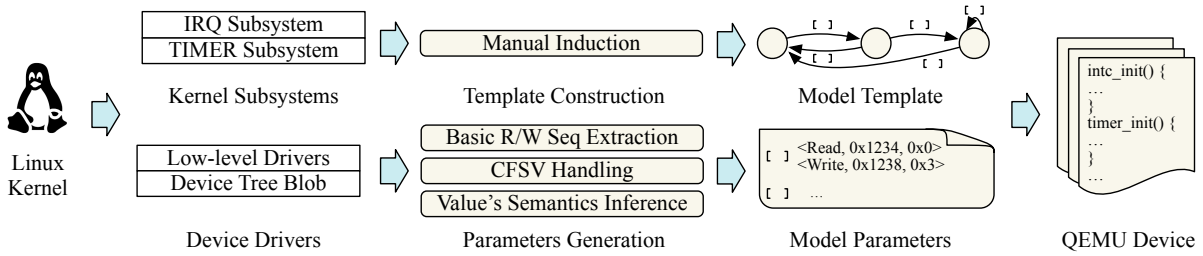


图 3.3 离线模型生成工作流示意图

统中手动建立其模型模板（小节 3.4.1），而模型参数则从低级驱动代码中自动生成（小节 3.4.2）。因此，为每个特定的外围设备生成模型就像一个填空的过程。最后，生成的模型可以指导（识别和控制）内核的执行，以成功地重新托管。我们开发了一个原型系统，命名为 FirmGuide。图 3.2显示了它的结构。它由两个部分组成：离线模型生成和在线内核启动。第一个组件分析 Linux 内核源代码以生成外围设备模型，而第二个组件使用合成的 QEMU 虚拟机重新托管嵌入式 Linux 内核。注意，尽管我们的系统在离线组件中需要 Linux 内核的源代码，但在在线内核启动中不需要源代码。节 3.4和节 3.5分别详细介绍了这两部分。

3.4 系统设计之离线模型生成

在离线模型生成中，我们用 Linux 内核的源代码半自动地生成中断控制器和定时器的外围设备模型。生成的外围设备模型（C 代码）可以用 QEMU 进行编译，以提供外围设备的全部功能。图 3.3显示了其过程。模型生成过程有两个部分：手动构建模型模

板和自动生成模型参数。

如图 3.3所示，模型模板是从 Linux 内核子系统中归纳出来的（小节 3.4.1）。每种类型的外围设备的模板都包含一个状态机，通过关键函数与 Linux 内核的子系统进行通信。换句话说，如果模型模板确切地知道 Linux 内核执行了哪一个关键函数，它就会做出正确的反应，转移其内部状态并采取下一步行动。模型模板实现了状态机的节点（状态）和单向边（状态转换表），但将转换条件留为空白（事件的触发）。请注意，模型模板的构建对每种类型的外围设备都是一次性的。

通过分析底层驱动代码而产生的模型参数，旨在描述过渡条件。请注意，触发这些事件的条件是执行关键函数的特定路径。例如，在图 3.1(a) 中，屏蔽中断源的事件意味着 irq_mask_callback() 被调用了。为了描述这些事件，我们结合三种方法来提取所有必要的信息：基本 R/W Seq 提取（小节 3.4.2），CFSV 处理（小节 3.4.3），以及值的语义推断（小节 3.4.4）。第一个方法生成了一个关键函数的执行路径的基本 R/W Seq。后两种方法补充了第一种方法，仅在 R/W Seq 不够用的情况下提取执行路径的额外属性。

3.4.1 手动构建模板

模型模板的构建是建立一个包含该类型外围设备的所有动作的状态机。为此，我们首先研究了 Linux 中对中断控制器和定时器子系统的抽象。具体来说，对于一个给定的外围设备，有三种事件可以被触发，它们来自 Linux 内核、其他连接的外围设备和外围设备本身。例如，一个中断控制器有一些事件，如屏蔽一个中断（来自 Linux 内核），请求触发一个中断（来自其他外围设备），以及触发一个中断（来自自身）。基于这些事件，我们设计了状态和状态转换表，但将转换条件（事件的触发）留为空白。注意，只有来自 Linux 内核的事件是作为空白处理的，而其他的事件触发条件很简单，比如提高或降低中断信号。我们为边缘触发和水平触发的中断控制器手工设计了 8 到 9 个事件和 4 到 5 个状态。类似的，我们为 clkevt 定时器设计的模型，具有 9 个事件和 4 个状态，为 clksrc 定时器设计的模型具有 4 个和 2 个状态。

图 3.4 展示了建模后中断控制器的工作流程。矩形是状态，单向黑色箭头是状态转换。双向的白色箭头是状态转移条件（即事件），它们是自动从特定的驱动程序中提取的。箭头上的标记是我们用来提取该过渡条件的关键函数。

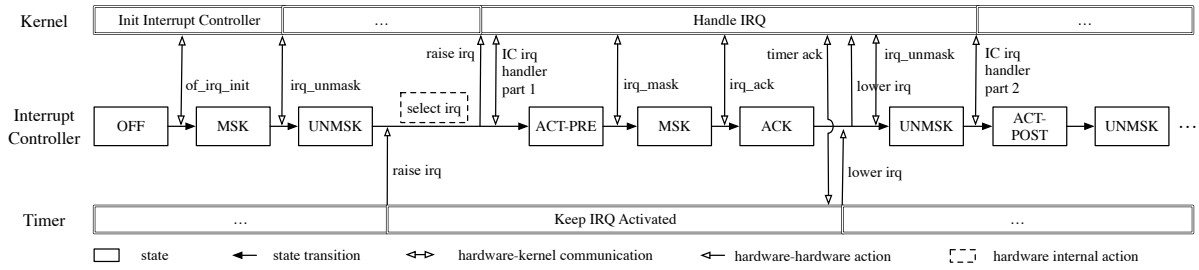


图 3.4 建模的中断控制器的工作流示意图

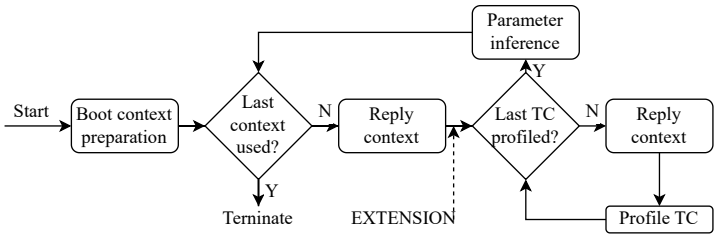


图 3.5 基本 R/W Seq 的提取流程图：TC 指状态转移条件；虚线箭头指小节 3.4.3中提到的扩展

3.4.2　自动提取基本 R/W Seq

基本 R/W Seq 是一个 MMIO 读或写操作的序列，用来表示一个关键函数的特定执行路径。事实上，一半以上的案例只使用基本 R/W Seq 来表示过渡条件。我们对关键函数应用符号执行，找出可以代表特定事件的 MMIO 读写序列的路径（例如，编号为 3 的中断源被屏蔽），这些读写序列被称为基本 R/W Seq。图 3.5显示了其一般工作流程。

启动上下文准备。将符号执行直接应用于一个关键函数需要一个执行环境。我们创建了一个简化的启动过程来获得相关内核子系统的初始上下文。具体来说，启动过程是由 Linux 内核的代码 start_kernel() 简化而来，但只对内核子系统进行必要的初始化（即只保留中断和时间子系统的初始化代码）。将符号执行应用到这个启动过程中，我们可能得到多个路径，都可以成功地完成整个启动过程。每条路径和它的内存状态（初始值）都是一个有效的启动上下文，因为在实践中多个硬件配置可以正确地启动 Linux 内核。初始值是在约束条件求解后得到的。此外，上下文的准备也顺便解决了 II-型外围设备的初始值的问题。需要强调的是，在符号执行中，我们为每个 MMIO 读操作引入一个新的符号，因为它确实是一个易失性操作。

重放上下文。在启动上下文准备之后，符号执行引擎（KLEE）已经在内存中保持了几个启动上下文。在执行关键函数之前，我们另外实现了一种上下文重放技术，通过用一个有效的启动上下文重新运行启动过程来重新创建这些启动上下文。由于 KLEE 只

使用一个主机 CPU 来进行状态探索，这样的重放技术也可以通过多核并行，充分利用主机 CPU 资源来加快分析速度。

转移条件获取。下一步是根据模型模板获得状态转移条件。例如，中断控制器的模型模板要求我们为每一个中断源（我们从设备树上获得中断源的列表）收集关键函数里蕴含的状态转移条件，这些关键函数有 irq_mask()，irq_ack()，irq_unmask() 等。每个状态转移条件都被记为一对 < 执行目标，关键函数 >。我们首先从状态机中准备好这些二元组，然后通过符号执行对带有执行目标的关键函数进行画像，来获得每个状态转移条件。我们用 “一次分析” 来表示这个对这个二元组列表的一次枚举。

- 设置执行目标：执行目标是描述状态转移条件的期望路径。每个关键函数可能有多条路径，而期望的路径是完成其真正功能的路径。例如，在图 3.1(b) 中，要找到 “触发编号为 3 的中断” 的执行路径，执行目标应该是这样的：它必须调用 generic_handle_irq() 一次，而且只有一次；它必须调用 generic_handle_irq() 并传递 3 作为第一个参数；它必须从函数中无误返回。
- 转移条件画像：路径中满足执行目标的 MMIO 操作序列被保存为画像文件。

参数推断。以下规则用于从画像文件中推断出基本 R/W Seq。

- 对于 MMIO 读，如果符号值有约束，我们会在基本 R/W Seq 列表中附加一个新节点 <MMIOR, addr, value>，其中的 value 是约束解算器提供的具体数值。这个节点告诉外围设备模型，当 Linux 内核从这个 MMIO 地址读取时，返回值为 value。
- 对于 MMIO 读，如果符号没有约束，我们会在基本的 R/W Seq 列表中附加一个新节点 <MMIOR, addr, USE_LAST_VALUE>。这个节点告诉外围设备模型，当从 MMIO 地址读出时，要提供写给它的最后一个值。
- 对于一个 MMIO 写，我们附加一个新的节点 code<MMIOW, addr, match(expr)>，其中 match() 是外围设备模型用来检查 MMIO 写入的值是否满足 expr。

3.4.3 自动处理 CFSV

基本 R/W Seq 在两个关键函数之间没有数据交换的情况下效果很好。根据我们的经验，这个假设大多数情况下是可行的。然而，数据交换可能发生在两个关键函数 之间。具体来说，它是在两个关键函数的执行中对非本地变量的读和写操作时发生的。一个函数的非本地变量是指生命周期长于该函数的变量，例如全局变量和堆变量。图 3.6展示

```
1 #define MASK_ADDR 0xFFFF0014
2 void irq_mask(u32 irq) {
3   u32 mask = 0x1 << irq;
4   /* The mask_cache is a CFSV, i.e.
5   * a non-local variable and a symbolic value. */
6   *mask_cache &= ~mask;
7   writel(*mask_cache, MASK_ADDR);
8 }
```

图 3.6 CFSV 示例

算法 1: CFSV 检测

input : $ctxt$, symbolic execution context
output: $CFSVs$, recognized CFSV set

```
1 init empty set CFSVs, nonlocal_rw;
2 do
3     state_merge_begin();
4     label known CFSVs in ctxt;
5     records ← do a pass of analysis using ctxt;
6     nonlocal_rw ← extract memory r/w info from records;
7     state_merge_end();
8     update CFSVs based on merged nonlocal_rw;
9 while CFSVs has increased;
```

了一个例子。第 6 行的 mask_cache 是一个堆变量。它记录了所有中断源的当前屏蔽状态。每当 Linux 内核调用 irq_mask() 或 irq_unmask() 时，它就会被更新，并且在第 7 行又被使用。这就导致了一个问题，即针对第 7 行画像的 expr 不能与实际情况相匹配，因为第 7 行中写的值可以改变。因此，不匹配的情况会发生，外围设备模型不能正常工作。

CFSV 定义。我们首先介绍一个概念，叫做 CFSV（Cross Function Symbolic Variable）。CFSV 是一个既可以读也可以写的非本地变量。由于 CFSV 的值可以在一个函数中更新并在另一个函数中使用，它的值取决于这些关键函数的执行和执行顺序。如果一个关键函数是无 CFSV 的，基本的 R/W Seq 就足以描述状态转移条件。相反，如果不是无 CFSV（读或写 CFSV），我们需要记录 CFSV 的读/写操作，通过添加 CFSV 信息扩展基本 R/W Seq，然后在外围设备模型中模拟 CFSV。

CFSV 检测。 算法 1描述了一种单调递增的寻找 CFSV 的算法。有几个假设可以保证算法最终能够收敛：在所有的关键函数中没有新分配的内存；没有符号指针；关键函数的大小合理，以便符号执行可以完成探索。这些假设保证了 CFSV 的数量是有限的和固定的。因此，算法 1中的 CFSV 是一个有上限的单调增加的集合。因此，该算法最终可以达到一个不动点。

请注意，第 3 行和第 7 行使用了 KLEE 中的一项技术，叫做状态合并[83]。我们对它做了拓展以支持 CFSV 相关信息的合并。合并状态有助于我们在一次分析中汇总 CFSV 的读/写操作，同时保持上下文的数量不变。第 4 行给上下文中已知的 CFSV 贴上标签，所有已知的 CFSV 都被视为假的 MMIO 寄存器，也就是说，每读一次 CFSV 就会引入一个新的符号。原因是 CFSV 在执行过程中可以被改变。因此，我们将其符号化为任何可能的值（过近似），以最大限度地提高路径探索的效果。在第 6 行，我们收集每个状态下所有非本地变量的读/写信息，然后合并这些信息。之后，我们用这些信息来更新 CFSV。图 3.5中的虚线箭头处加入了 CFSV 检测。

为了处理 CFSV，我们在虚拟外设模型中模拟其数据传播。与 CFSV 检测类似，我们在分析过程中把 CFSV 当作一个假的 MMIO 寄存器。引入的符号在配置文件中被标记为 CFSV 标签。然后，我们通过扩展基本的 R/W Seq 与两种新类型的节点，称为 <CFSVR, addr, val> 和 <CFSVW, addr, expr>，在参数推断中记录其数据传播。最后，我们分配了全局变量，并在外围设备模型中模拟了 CFSV 和基本 R/W Seq 的数据传播。

3.4.4 自动推断 MMIO 值的语义

以上两种方法解决了描述关键函数的路径的问题。然而，为了仿真一个定时器，模型需要理解 MMIO 寄存器的值的语义，以便对一些关键函数做出正确的反应。一种情况是，我们的定时器模型需要理解 Linux 内核何时要收到下一个定时器中断。首先，Linux 内核确定了一个以 cycle_t 为单位的值（定时器设备最小的单位），然后它将这个单位与特定的定时器设备进行换算，单位转换的公式可以在符号执行过程中收集。

然而，从定时器模型的角度来看，理解 Linux 内核写入的值所代表的时间段需要我们将该值从定时器设备单位转化为回 cycle_t。这意味着我们需要对公式进行反向操作。大多数情况下，转换公式就像 $y = k * x$ 一样简单，其中 k 是一个常数，x, y 是用两个单位表示的时间段。我们可以直接检测这些情况，用轻量级的方式处理它们（硬编码其反函数）。如果我们遇到复杂的转换公式，我们需要在外围设备模型中使用约束求解器。我们在实验过程中没有遇到复杂的公式。

我们再次通过添加单位转换信息来扩展我们的基本 R/W Seq。这个扩展有助于定时器外围设备提供准确的时间模拟。

3.4.5 实现细节

在离线模型生成中，模型模板有 2812 行 C 代码（中断控制器 1712 行，定时器 1100 行）。参数生成部分基于 KLEE，包括用于静态分析的 4869 行 C 代码，用于调整 KLEE 的 1902 行 C++ 代码，以及 1110 行的充当胶水的 Python 代码。

源代码的预处理。FirmGuide 利用 LLVM 和 KLEE[84]来对 Linux 内核源代码进行静态分析。我们对代码进行了预处理，原因有三。我们用等价的 C 函数替换内联汇编，因为 LLVM 不能分析汇编代码；为了使分析更容易，我们通过修改头文件，简化了一些库的函数，并将静态变量改为对其他模块可见，这些改动都是一次性的；此外，因为这些针对简短的函数和有限的变量并没有改变代码的语义，并不影响静态分析的结果。

基于符号执行引擎的分析。我们的符号分析是在 KLEE 基础上开发的。输入是目标 Linux 内核的源代码以及它的设备树。输出是生成的模型参数。这些参数将被应用在模型模板中，以生成可以直接编译的外围设备模型。详细来说，我们将目标 Linux 内核源代码编译为一个链接的 LLVM IR 文件，并在 IR 文件上采取以下策略运行符号执行。

- 通过链接的 IR 文件的假入口点来控制符号执行的流程。我们将自定义的 main() 函数添加到 LLVM 的 IR 文件中，让符号执行从这里开始。在 main() 函数中，首先会执行简化的内核启动过程（例如，time_init(), init_IRQ()）以获得启动环境，然后启动分析代码以生成模板参数（图 3.5）。
- 通过链接我们定制的实现而不是原始代码，覆盖整个 Linux 内核的使用的通用函数。我们定制的库涵盖了 Linux 内核的中断子系统、时间子系统、设备树库、内存管理和 stdlib。这简化了符号执行，避免了可能的路径爆炸。
- 通过扩展 KLEE 的 SpecialFunctionHandler 接口，增加便于分析的辅助功能。

3.5 系统设计之在线内核启动

在线内核启动组件是通过几个现有的工具实现的。给定一个固件镜像，我们首先使用 Binwalk 从镜像文件中提取出 Linux 内核和设备树。然后我们从设备树中提取外围设备列表。对于列表中的每个外围设备，我们使用设备树中的 compatible 属性来匹配从离线模型生成中的获得的外围设备模型。这是可行的，因为我们通过相同的 compatible 属性来组织生成的外围设备模型。外围设备模型（C 代码）将与 QEMU 一起被编译。为了

表 3.1 代表性的 OpenWrt 子目标构成的固件数据集

Subtarget	Source Code		Firmware			
	Rev. of OpenWrt	Ver. of LinuxKern	Archi-tecture	# of Firmware	# of SoC	# of Vendors
ramips/rt305x	15.05	3.18.20	mipseb	5249	4	55
ath79/generic	19.07.1	4.14.167	mipsel	613	15	24
kirkwood/generic	15.05	3.18.20	armel	482	3	6
bcm53xx/generic	15.05	3.18.20	armel	388	3	1
oxnas/generic	15.05	3.18.20	armel	176	1	4
Summary	×2	×2	×3	6908 个	26 个	90 个

表 3.2 离线模型生成的结果

Subtarget	Interrupt Controller	Timer	# of Paths	# of Solutions	First/All Solution (s)	Exists CFSV (y/n)	Timer Semantics	LoC
ramips/rt305x	ralink-rt2880-intc	not necessary	262	4	1/2	n	-	3,366
ath79/generic	qca,ar7240-intc	not necessary	110,083	1,134	5/943	n	-	4,138
kirkwood/generic	marvell,orion-intc marvell,orion-bridge-intc	marvell,orion-timer	132	2	2/3	y	$y = \sim x$	4,790
bcm53xx/generic	arm,cortex-a9-gic	arm,cortex-a9-global-timer arm,cortex-a9-twd-timer	150,336	2,592	2,027/24,070	y	$y = x_1 << 32 + x_2$	3,537
oxnas/generic	arm,arm11mp-gic	arm,arm11mp-twd-timer plxtech,nas782x-rps-timer	52,332	1,246	914/16,184	y	$y = x$	3,366

在 QEMU 中添加一个新的虚拟机，我们需要用 C 语言编写一个新的机器文件，初始化所有的外围设备。我们自动生成这个通用的机器文件，最后用生成的 QEMU 虚拟机来重新托管 Linux 内核以及由 Buildroot 生成的准备好的 RAM 文件系统作为根文件系统。整个在线内核启动组件是用 Python 写的，有 5519 行代码。

3.6 实验验证

3.6.1 实验设置

为了评估我们系统的有效性，我们使用从 OpenWrt 下载的覆盖多个片上系统的固件镜像进行实验。一个系列的片上系统的源代码被组织在一个子目标中。我们按照以下三个标准选择了前五个子目标：它们支持设备树；它们覆盖不同的架构和编码；它们有许多发布的固件镜像而受欢迎[78-79]。表 3.1显示了我们数据集的细节。由于 OpenWrt 为 ramips/rt305x 发布了更多的固件镜像文件，所以 ramips/rt305x 的数量要比其他的多。请注意，我们在 OpenWrt 上进行实验是因为这个数据集涵盖了不同的片上系统、供应商、架构和内核版本，数据集具有代表性。我们在一台有两个 Intel Xeon Silver 4114 的处理器、128GB 内存、Ubuntu 16.04.6 LTS 系统的服务器上进行了所有实验。

表 3.3 模拟的所有 II-型外围设备中，初始值不为零的 II-型外围设备的比例结果

Subtarget	ramips/ rt305x	ath79/ generic	kirkwood/ generic	bcm53xx/ generic	oxnas/ generic
Count	1/10	2/15	3/26	2/4	2/9

3.6.2 离线模型生成评估

模型参数生成。如表 3.2所示，FirmGuide 生成了 I-型外围设备的模型参数。请注意，我们没有为两个 MIPS 子目标中的定时器（标记为 not necessary）生成参数。因为 QEMU 的 MIPS 处理器已经实现了这些定时器。具体来说，第 4 列显示符号执行引擎在模型生成过程中探索的路径数量。第 5 列显示了我们的系统为成功重新托管找到的最终解决方案的数量。第 6 栏详细介绍了获得第一个解决方案和所有解决方案的时间。找到第一个解决方案的最长时间在一小时之内，显示出我们的方法比手动开发外围模型要快。第 7 栏给出是否存在 CFSV。第 8 列给出了该子目标的定时器中使用的单位转换公式。x 代表与特定定时器的单位一致的时间间隔，y 是以 cycle_t 为单位的值。我们用 x_1、x_2 代表外围设备用来存储时间的两个寄存器。最后一列列出了生成外围设备模型的代码行。

表 3.4 在线内核启动的结果：Unpack 指解压成功的固件镜像的数量，Kernel 指检测到嵌入式 Linux 内核的数量，User Space 指进入用户空间的嵌入式 Linux 内核的数量，Shell 指弹出控制台窗口的嵌入式 Linux 内核的数量

SoC	Unpack	Kernel	Booting Validation	
			User Space	Shell
Ralink RT3050	1164	1164	1144 (98.28%)	1052 (90.38%)
Ralink RT3052	1815	1815	1815 (100.00%)	1661 (91.52%)
Ralink RT3352	173	173	173 (100.00%)	157 (90.75%)
Ralink RT5350	1632	1632	1611 (98.71%)	1475 (90.38%)
subtarget: ramips/rt305x	4784	4784	4743 (99.14%)	4345 (90.82%)
Atheros AR7161	36	36	20 (55.56%)	20 (55.56%)
Atheros AR7241	20	20	12 (60.00%)	12 (60.00%)
Atheros AR7242	24	24	24 (100.00%)	24 (100.00%)
Atheros AR9330	4	4	4 (100.00%)	4 (100.00%)
Atheros AR9331	24	24	12 (50.00%)	12 (50.00%)
Atheros AR9341	10	10	4 (40.00%)	4 (40.00%)

表 3.4 续上页

Atheros AR9342	24	24	24 100.00%)	24 (100.00%)
Atheros AR9344	70	70	64 (91.43%)	64 (91.43%)
Qualcomm Atheros QCA9531	22	22	16 (72.73%)	16 (72.73%)
Qualcomm Atheros QCA9533	41	41	14 (34.15%)	14 (34.15%)
Qualcomm Atheros QCA9557	64	64	64 (100.00%)	64 (100.00%)
Qualcomm Atheros QCA9558	54	54	50 (92.59%)	50 (92.59%)
Qualcomm Atheros QCA9560	16	16	16 (100.00%)	16 (100.00%)
Qualcomm Atheros QCA9561	18	18	14 (77.78%)	14 (77.78%)
Qualcomm Atheros QCA9563	114	114	106 (92.98%)	106 (92.98%)
subtarget: ath79/generic	541	541	444 (82.07%)	444 (82.07%)
Broadcom BCM4708A0	241	241	241 (100.00%)	241 (100.00%)
Broadcom BCM4709A0	128	128	128 (100.00%)	128 (100.00%)
Broadcom BCM47189	19	19	19 (100.00%)	19 (100.00%)
subtarget: bcm53xx/generic	388	388	388 (100%)	388 (100%)
Marvell 88F6192	20	20	20 (100.00%)	20 (100.00%)
Marvell 88F6281	208	204	204 (100.00%)	144 (70.59%)
Marvell 88F6282	102	102	100 (98.04%)	80 (78.43%)
subtarget: kirkwood/generic	330	326	324 (99.39%)	244 (74.85%) †
PLX NAS7820	149	149	48 (32.21%)	48 (32.21%)
subtarget: oxnas/generic	149	149	48 (32.21%)	48 (32.21%) ✦
Overall	6,192	6,188	5947 (96.11%)	5469 (88.38%)

II-型外围设备。对于 II-型外围设备，我们自动生成没有状态机的外围设备模型，但有适当的寄存器的初始值。这些值是在启动环境准备中计算的（小节 3.4.2）。表 3.3 显示了模拟的 II-型外围设备的数量和那些具有非零初始值的外围设备。在 64 个外围设备中，总共有 10 个 II-型外围设备的初始值不为零。

3.6.3 在线内核启动评估

除了离线模型生成，我们还进行了实验来重新托管一些固件镜像文件。实验结果显示，我们生成的 QEMU 机器可以成功地重新托管超过 95% 的 Linux 内核。在评估中，我

们通过解析 QEMU 调试选项-d cpu 所提供的代码追踪来判断 Linux 内核镜像是否已经切换到用户空间；我们还通过检测启动过程中输出的信息，即 Welcome to Buildroot, 来检查是否已经弹出了一个控制台窗口。表 3.4 展示了总体的结果。总的来说，6908 个固件中，成功解包 6192 个，获得了 6188 个 Linux 内核。其中，5947 (96.11%) 进入了用户空间，而 5469 (88.38%) 成功弹出了控制台窗口。我们手动分析了失败案例的原因。

触发了一个两次释放安全缺陷†。kirkwood/generic 中的一些 Linux 内核在函数 orion-_nand_probe()[85]中存在一个两次释放安全缺陷，如果闪存设备不存在，那么 clk_put() 会被调用两次。这个安全缺陷存在于 4.9 之前的 Linux 内核中。有趣的是，当闪存设备是正常的，这个安全缺陷不会被触发；然而，它在 FirmGuide 中被触发了，因为我们在模拟器中只有一个无功能的未初始化的闪存设备（可看作是恶意的闪存设备）。

不支持根文件系统✦。oxnas/generic 中的一些 Linux 内核不支持 Linux 内核默认支持的 ramfs。我们推测删除对这个文件系统的支持是为了减少镜像的大小。

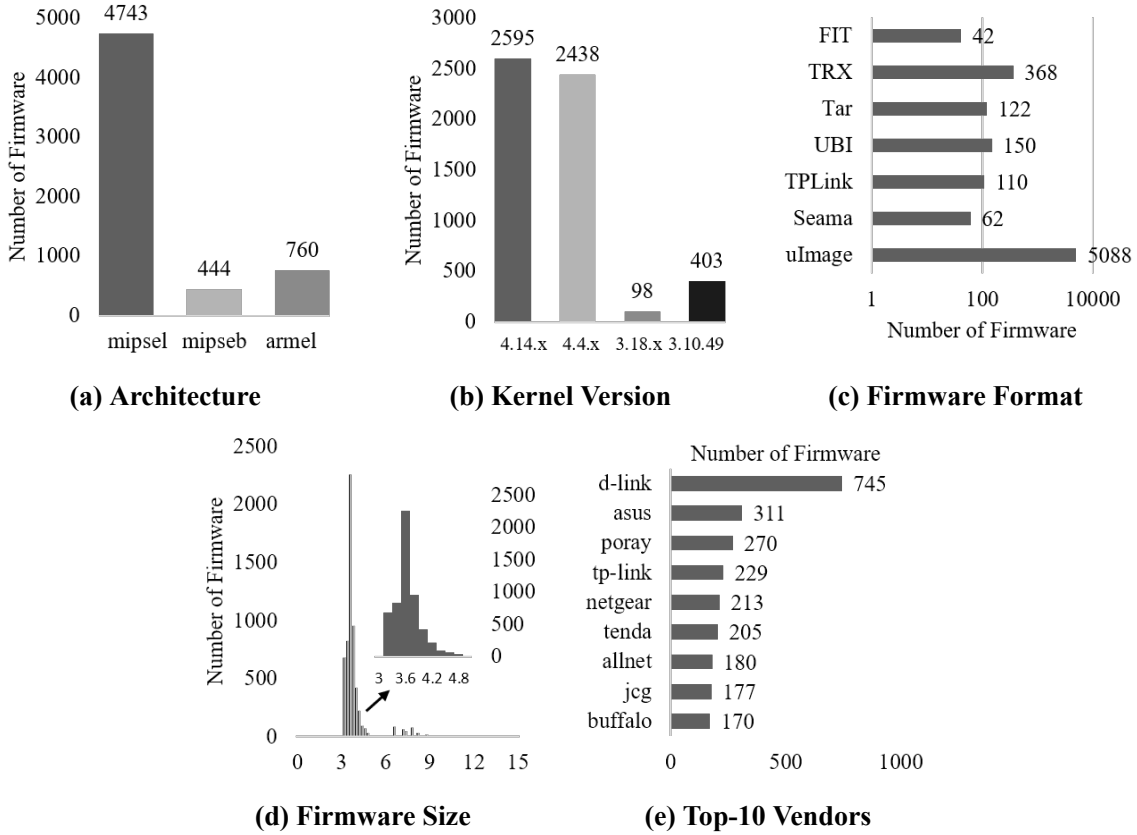
3.6.4 固件多样性评估

我们通过探索重新托管的 Linux 内核的多样性，进一步研究 FirmGuide 的可扩展性。图 3.7显示，FirmGuide 可以重新托管不同的嵌入式 Linux 内核。

- 架构。 FirmGuide 支持小端编码的 ARM32、大端编码的 MIPS32 和小端编码的 MIPS32 的 Linux 内核镜像（图 3.7a）。FirmGuide 是独立于架构的。
- 内核版本。 FirmGuide 可以重新托管 4 个主要版本和 22 个不同的次要版本的嵌入式 Linux 内核，表明生成的模型对特定的内核版本没有要求（图 3.7b）。
- 固件格式。我们扩展了 Binwalk 以支持图 3.7c中列出的七种固件格式。其中，传统的 uImage 是最流行的固件格式。
- 固件大小。支持的固件大小从 3MB 到 16MB 不等，平均约为 3.6MB（图 3.7d）。
- 片上系统和供应商。如表 3.4所示，FirmGuide 已支持 26 个片上系统。在图 3.7e中，我们列出了前十名供应商的固件镜像的数量。

3.6.5 重新托管的内核功能性评估

我们进行了两个实验来证明由 FirmGuide 启动的固件的功能。首先，我们使用 LTP 的系统调用测试来测试这些启动的固件图像。LTP 包含文件系统、I/O、内存管理、调度



(a) Architecture　　**(b) Kernel Version**　　**(c) Firmware Format**

(d) Firmware Size　　**(e) Top-10 Vendors**

图 3.7 FirmGuide 支持不同的架构、Linux 内核版本、固件格式、固件大小（MB）和固件供应商：在图 3.7b中，4.14.x 有 8 个子版本，4.4.x 有 11 个，3.18.x 有 2 个子版本，总共有 22 个不同版本

器等的测试。总的来说，在所有 1259 个系统调用测试中，1049 个通过了，164 个因没有引入被测试的内核版本中被跳过了，还有 46 个测试失败。我们分析了失败的测试，并总结了失败的原因（表 3.5）。大多数失败的原因是假网络设备或未实现的系统调用（该固件中包含的 Linux 内核不支持该系统调用）。

第二，我们比较了由 FirmGuide 生成的外围设备与手动编写的外围设备的功能。我们比较了 plxtech,nas7820 设备。为了手工编写外围设备的模拟代码，我们首先学习 I-型外围设备的驱动源代码，然后根据人的理解编写 QEMU 代码。为 plextec,nas7820 设备编写 I-型外围设备的模拟代码的成本约为 1 周/人（假设已经学习了驱动开发，且成本时间包括代码开发和调试）。我们利用手工开发的外围设备模型进行了测试，表 3.6显示了结果，其中 Ground Truth 代表手动编写的虚拟外设代码。结果显示，生成的具有与手工编写的虚拟外设相同的功能。

表 3.5 系统调用测试失败的原因汇总

Reason	Count
Bugs or vulnerabilities of the Linux kernel are not patched	6
Network device is not available	14
Some syscalls are not implemented	20
Other	6
Total	46

表 3.6 手工编写与自动化生成虚拟外设的系统调用测试的结果

Models	Pass	Skipped	Failed	Total
Generated	1049	164	46	1259
Ground Truth	1049	164	46	1259

3.6.6 模型引导内核执行有效性评估

对有效性的威胁来自三个方面。

- 在我们的方法中，模型模板应该由人类专家手动建立。如果一些复杂的 II-型外围设备的状态机过于复杂，这个人工过程可能会成为瓶颈。
- 为了将该方法移植到相同或其他操作系统中的相同或其他外围设备上，我们的方法需要在目标操作系统的合适层上有一个外围设备接口。如果该层是一个通用的接口，其功能有明确的外围设备语义，那么该层是合适的。否则，R/W Seq 不能有效地指导（识别和控制）内核的执行。
- 我们使用符号执行来寻找可用的启动环境和 R/W Seq。如果关键函数非常复杂，那么参数推断的过程因路径爆炸而失败。然而，从我们支持 Linux 的经验来看，因为关键函数往往是简单的，路径爆炸并不是一个威胁。此外，我们在节 3.4中的上下文重放技术可以使生成过程并行运行（KLEE 本身不支持并行运行）。此外，我们不需要为一个给定的源代码找到所有的解决方案，因为为启动过程找到一个满意的路径和为每个状态转换条件找到一个 R/W Seq 就足够了。

3.7 安全应用

我们在 FirmGuide 上部署了两个安全应用。这些应用是示范性的，不是我们的主要贡献。其他基于 QEMU 的工具[86-89]，例如 S2E[90]，也可以应用。

表 3.7 对重新托管的嵌入式 Linux 内核的漏洞测试结果：✓表示成功触发，†表示成功利用，而✗表示失败；其他三个符号（□ ◊ △）代表 Linux 内核的不同版本（□ 3.10.49, ◊ 3.18.20, △ 4.4.42）

CVE ID	CVE Type	Status	Version
CVE-2016-5195	Race Condition	✗	N/A
CVE-2016-8655	Race Condition	✓ †	□
CVE-2016-9793	Integer Overflow	✓	□ ◊
CVE-2017-7038	Integer Overflow	✓ †	◊
CVE-2017-1000112	Buffer Overflow	✓ †	△
CVE-2018-5333	NULL Pointer Dereference	✓ †	□ ◊

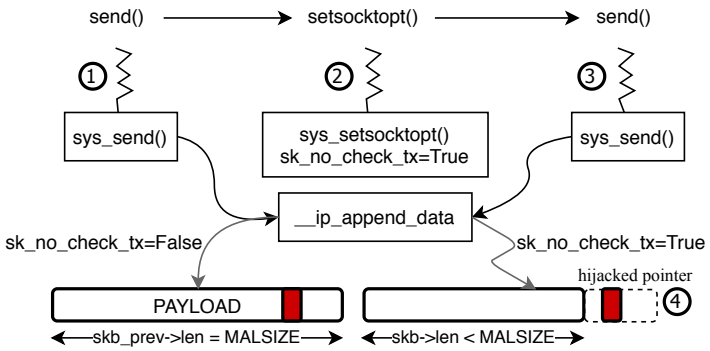


图 3.8 CVE-2017-1000112 的漏洞利用过程示意图

3.7.1　嵌入式 Linux 内核漏洞分析

我们首先收集了 6 个 Linux 内核漏洞（表 3.7），针对一个带有 plxtech,nas7820 设备的 ARM 的片上系统的 Linux 内核，对它的几个不同的版本进行了分析，我们使用 FirmGuide 来重新托管这些嵌入式 Linux 内核。接下来，利用 QEMU 的调试能力，我们成功地触发了 5 个漏洞，开发了 4 个内核漏洞利用，并分析了失败案例的原因。

漏洞触发。在用 QEMU 的调试功能检查这个新的执行环境后，5 个漏洞被成功触发，为进一步利用奠定了基础。CVE-2016-5195（又名 DirtyCow）无法被触发的原因是目标 Linux 内核不支持 madvise 系统调用，而这是触发竞争条件的必要条件。

漏洞的理解和利用。对于重新托管的有漏洞的 Linux 内核，我们继续使用 QEMU 的调试功能来理解漏洞，搜索可利用的函数指针，成功开发 4 个漏洞利用。例如，CVE-2017-1000112[91]，一个缓冲区溢出漏洞，当数据包处理程序从 UFO（UDP 片段卸载）路径切换到非 UFO 路径时，就会被触发。为了利用它，skb_prev->len 和被劫持的指针的偏移量必须被仔细计算。我们使用 QEMU 的调试功能寻找合适的溢出大小。

详细地利用过程如图 3.8所示。第一步，调用 send 系统调用，其载荷为 MALSIZE 以及 MSG_MORE 标志，目的是将随后的数据包处理流程从 ___ip_append_data() 引导

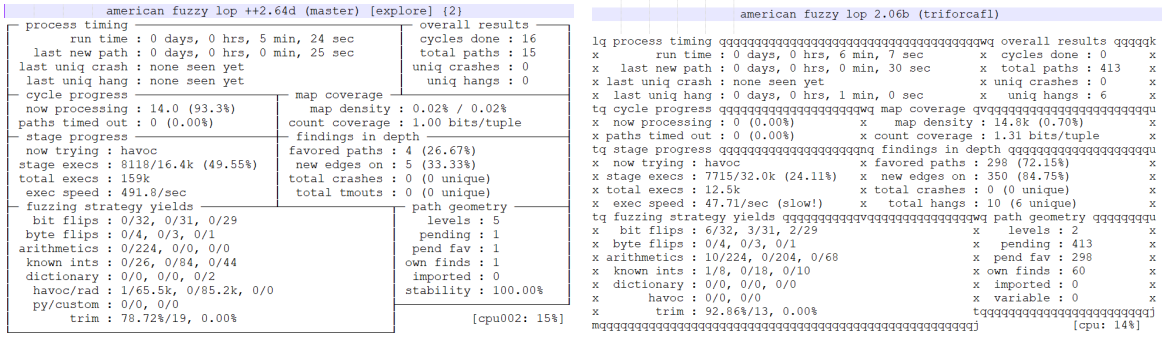
```
Breakpoint 1, __ip_append_data (sk=0xcfb12000, fl4=0xcfb12240, queue=0xcfb120e0, cork=0xcfb12220, pfrag=0xcf9bf7c0,
    getfrag=0xc02e6048 <ip_generic_getfrag>, from=0xcfb37ed0, length=3612, transhdrlen=8, flags=32768)
    at net/ipv4/ip_output.c:880
880     struct inet_sock *inet = inet_sk(sk);
(gdb) info registers
r0             0xcfb12000   -810475520
r1             0xcfb12240   -810474944
r2             0xcfb120e0   -810475296
r3             0xcfb12000   -810475520
r4             0xcfb12000   -810475520
r5             0xcfb120e0   -810475296
r6             0x8  8
r7             0xcfb12220   -810474976
r8             0xcfb12240   -810474944
r9             0xc02e6048   -1070702520
r10            0xcfb37ed0   -810320176
r11            0xcfb37d8c   -810320500
r12            0xcfb120e0   -810475296
sp             0xcfb37c18   0xcfb37c18
lr             0xc02e8194   -1070693996
pc             0xc02e611c   0xc02e611c <__ip_append_data+40>
cpsr           0x13 19
(gdb) x/8i $pc
=> 0xc02e611c <__ip_append_data+40>:    str r3, [r11, #-292]    ; 0xfffffedc
   0xc02e6120 <__ip_append_data+44>:    ldr r3, [r11, #-364]    ; 0xfffffe94
   0xc02e6124 <__ip_append_data+48>:    ldr r3, [r3, #8]
   0xc02e6128 <__ip_append_data+52>:    str r3, [r11, #-288]    ; 0xfffffee0
   0xc02e612c <__ip_append_data+56>:    mov r3, #0
   0xc02e6130 <__ip_append_data+60>:    str r3, [r11, #-328]    ; 0xfffffeb8
   0xc02e6134 <__ip_append_data+64>:    mov r3, #0
   0xc02e6138 <__ip_append_data+68>:    str r3, [r11, #-324]    ; 0xfffffebc
(gdb) info breakpoints
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0xc02e611c in __ip_append_data at net/ipv4/ip_output.c:880
    breakpoint already hit 1 time
(gdb)
```

图 3.9 在重新托管的 Linux 内核 4.4.42 中调试 CVE-2017-1000112 示意图

到 ip_ufo_append_data()。调用 setsockopt 系统调用来设置 sk_no_check_tx。调用 send 系统调用来设置 sk_no_check_tx。由于 sk_no_check_tx 被设置，___ip_append_data() 中的错误条件检查将处理从 ip_ufo_append_data()（UFO 路径）切换到本地主循环（非 UFO 路径）。由于由 MALSIZE 控制的 skb_prev->len 大于非 UFO 处理所能提供的大小，在 skb_copy_and_csum_bits() 中存在堆溢出，恶意的 PAYLOAD 就可以劫持一个函数指针。

3.7.2　嵌入式 Linux 内核模糊测试

我们还移植了两个模糊工具，TriforceAFL[82]和 UnicoreFuzz[81]。TriforceAFL 被用来随机地调用嵌入式 Linux 内核中的所有系统调用。相反，UnicoreFuzz 是一种基于模拟器的模糊测试方法，它基于代码覆盖率反馈，被用来模糊测试内核空间中的 II-型外围设备驱动程序的解析部分，图 3.10展示了模糊测试的状态。即使没有新的漏洞，FirmGuide 仍然对 Linux 内核的模糊测试做出了贡献。FirmGuide 并不专注于任何特定的模糊技术，而是提供一个基础设施来分析嵌入式 Linux 内核。其次，有了 FirmGuide，只需付出少量努力，基于 QEMU 的模糊测试工具可以应用于测试嵌入式 Linux 内核。



(a) Unicorefuzz　　**(b) Trifore Linux syscall fuzzer**

图 3.10 内核模糊测试示意图

3.8　本章小结

在本章中，我们提出了一种叫做模型引导的内核执行的新技术，以重新托管嵌入式固件的 Linux 内核。它利用内核对外围设备和内核外围设备交互的抽象，以半自动化生成有状态的外围设备模型。然后，生成的模型可以用来合成 QEMU 虚拟机来重新托管嵌入式 Linux 内核。我们实现了一个名为 FirmGuide 的原型系统。它生成了具有完整功能的 9 个外围设备模型和具有最小功能的 64 个模型，涵盖了 26 个片上系统。我们用从互联网上下载的固件镜像进行的评估表明，它可以成功地重新托管超过 95% 的 Linux 内核，涵盖了两个架构和 22 个内核版本。两个安全应用已经被应用在重新托管的内核上，以证明 FirmGuide 可以为嵌入式 Linux 内核建立动态分析工具的基础。

4 基于依赖感知消息模型的虚拟执行环境模糊测试研究

Linux 物联网设备虚拟化的第二个核心目标是维护虚拟执行环境的安全性。因为众多且复杂的虚拟 Linux 外设是虚拟执行环境最大的攻击面，所以实现高安全性的虚拟执行环境需要通过高效的测试得到高安全的虚拟 Linux 外设，从虚拟执行环境的测试出发在开发阶段保证高安全性。因此，本章提出了基于依赖感知消息模型的新技术，通过模糊测试虚拟 Linux 外设加强整个虚拟执行环境的安全性。

4.1 引言

虚拟机管理程序（Hypervisor）或虚拟机监控器（Virtual Machine Monitor，简称 VMM）被广泛部署在云基础设施中。它们通过由 I/O 操作（Port Mapped I/O，简称 PIO 或 Memeory Mapped I/O，简称 MMIO）驱动的虚拟外设，将数据和指令从客户操作系统传输到主机环境。这些 I/O 操作遵循特定的协议，因此被称为虚拟外设消息。

虚拟外设是虚拟机管理程序中最大的攻击面。虚拟机管理程序将不受信任的客户虚拟机与虚拟机管理程序和所有其他虚拟机隔离。其中一个关键的安全属性是，客户不能从其客户虚拟机中逃逸。然而，虚拟机管理程序是复杂的软件，研究人员已经发现了逃逸它们的方法，这些逃逸是由于虚拟外设中的漏洞造成的[92]。根据我们的 CVE 调查[93]，QEMU 中的漏洞有 57.4% (252/439) 是在虚拟外设中发现的。

虚拟外设的安全性一直受到严格的检查[50-55,94-95]。自 2017 年 VFD[95]以来，模糊测试已经成为主导方法，因为它通过具体的执行隐含地抽象了设备的复杂性，优于符号执行和静态分析。后来，一个独立于平台的黑盒模糊测试器[53]由于其高吞吐量和多维输入而发现了多个安全缺陷。当开始考虑覆盖率反馈[54]和通过 DMA 通道提供的客户机数据[49,54-55]时，虚拟外设的模糊测试技术得到进一步推进。

尽管已发现数百虚拟外设漏洞，但由于两个挑战被忽视，现有解决方案仍然有限。

消息内依赖性：一个虚拟外设消息中的一个字段可能依赖于另一个字段。客户机通过虚拟外设消息与虚拟外设进行通信。每个虚拟外设消息都遵循一个给定的消息结构，并对具有不同语义的消息字段进行编码。例如，一个四字节的标量或一个指针。特别的

是，一个字段可能依赖于另一个字段。例如，数据字段中的一个比特位可以告诉虚拟外设指针字段的类型。没有意识到这种依赖关系的虚拟外设模糊测试器在到达某些代码时速度较慢，甚至可能会错过关键功能。

消息间依赖性：一个虚拟外设消息可能依赖于之前发出的消息。一个虚拟外设消息可以修改一个虚拟外设的内部状态，几条消息可以连锁起来形成一连串复杂的交互。在虚拟外设中，两个消息可能路径不同，但被设备内部状态纠缠在一起，说明消息是有序序列。不知道这些依赖关系的变异策略可能会违反顺序约束，浪费了时间和硬件资源。

现有的解决方案已经显示了虚拟外设模糊测试的优点，但因为没有意识到以下两个挑战，所以可扩展性或效率有限。

可扩展性。一个可扩展的虚拟外设模糊测试器需要较少的手动操作和灵活的系统设计，以支持不同的虚拟机管理程序、体系结构和虚拟外设类别。Nyx[54]介绍了基于同一前端的两种配置：Nyx-Legacy 和 Nyx-Spec（如果一个参数同时适用于 Nyx-Legacy 和 Nyx-Spec，我们就只提 Nyx）。与前者只涉及常规的 PIO/MMIO 操作相比，后者增加了一个手动编码的数据结构来支持 DMA 操作，例如，为 XHCI 分配链表。具体来说，在处理客户虚拟机提供的 DMA 数据时，Nyx-Spec 需要人为地编码给定的虚拟外设的规范，能迅速实现高代码覆盖率，但若添加新的虚拟外设，则涉及大量的人力。

效率。一个高效的虚拟外设模糊测试器必须快速提高代码覆盖率。除了考虑规范以加快代码覆盖率的提高外，Nyx, V-Shuttle[55]和 MorPhuzz[49]还错过了其他机会，因为它们的突变粒度太大或者太小。特别是，Nyx 使用了一个通用图，在每个节点中编码多个虚拟外设消息。由于它避免了节点间的交叉突变，这种方法限制了 Nyx 对更多交错行为的探索。相反，V-Shuttle 和 MorPhuzz 随机地突变字节，忽略了它们的结构表示，打破了两个连续消息的语义。

我们的目标是克服这两个挑战。在对虚拟外设进行模糊测试时，基于以下两个观察，实现这两个方面：可扩展性和效率。首先，我们注意到，源代码对消息语义进行编码，作为消息结构的参考。广泛使用的虚拟机管理程序（QEMU 和 VirtualBox）是开源的，编码了如何与虚拟外设互动的丰富信息。此外，源代码适合于自动分析，而且与复杂的规范相比，分析的劳动强度较低。其次，格式良好的信息可以覆盖更多的代码，并为模糊测试器提供更好的反馈，以便将来进行突变。

我们引入了一个新的依赖感知的虚拟外设模糊测试框架 ViDeZZo（Virtual Device

Fuzzer），它同时考虑了消息内和消息间的依赖。

轻量的消息内标记。为了支持消息内部的依赖性，我们设计了一个新颖的、轻量级的描述性语法（小节 4.3.1）。在审查源代码时，虚拟外设的安全分析师可以用我们的描述性语法记录消息内注释，以使模糊测试器知道如何处理消息内的依赖关系。我们认为，在 Nyx-Spec 中使用的来自硬件规范的完整语法和 V-Shuttle 和 MorPhuzz 中使用的基于启发式的方法之间，我们的轻量级语法是一个很好的折中。我们对注释的提取进行了半自动化处理。这里的低人工工作量支持可扩展性。

新颖的消息间突变器。为了处理消息间的依赖性，我们基于虚拟外设消息作为突变原子，设计了三类新的突变器。这些突变器创建单一消息或形成消息序列，利用模糊测试的遗传特性提供一致性（消息级）、多样性（序列级）和语义（组级）（小节 4.3.2）。这些消息感知突变器不仅能自我学习消息间的依赖，还能保持不同突变颗粒度的优势。

基于以上两种技术，我们在节 4.4中介绍了 ViDeZZo 的设计。ViDeZZo 有两个部分：ViDeZZo-Core 和 ViDeZZo-VMM 的绑定。前者管理模糊测试的输入，将其解析为虚拟外设消息，并根据我们的设计处理这些信息。后者，即 ViDeZZo-VMM，注册目标虚拟外设，在不运行任何操作系统的情况下初始化客户虚拟机，并分发特定虚拟机管理程序的消息。ViDeZZo-Core 与虚拟机管理程序无关，而 ViDeZZo-VMM 需要为每个新的虚拟机管理程序进行定制。灵活的系统设计使 ViDeZZo 的可扩展性得以实现。

CodeQL 推理引擎会自动推断消息内依赖注释，分析者可以对其进行调整，详见小节 4.5.1。小节 4.5.2和小节 4.5.3 分别总结了 ViDeZZo-Core 和 VideZZo-VMM 的实现。

重要的是，ViDeZZo-Core 实现了持久化模式，避免了耗时的分叉以提高性能。我们利用反射性的差分调试来解决由于累积的内部状态而产生的副作用。具体来说，ViDeZZo 存储所有的中间测试案例，并支持差分调试[96]，将收集的种子减少到最小的稳定的概念验证程序（Proof of Concept，简称 PoC）。

与以前的工作相比，ViDeZZo 既可扩展又高效。ViDeZZo 目前支持两个虚拟机管理程序，即 QEMU 和 VirtualBox，四个架构，即 i386，x86_64，AArch32 和 AArch64，28 个虚拟外设，五个虚拟外设类别，即 USB，net，display，audio 和 storage，并且更快地达到可比的代码覆盖率。ViDeZZo 在漏洞挖掘方面也很有效。我们成功地复现了 24 个现有的错误，并发现了 28 个新的错误，包括多样的漏洞类型，到目前为止获得了一个 CVE。我们一直在积极与 QEMU 和 VirtualBox 社区合作，并提供了 7 个已接受的补丁。

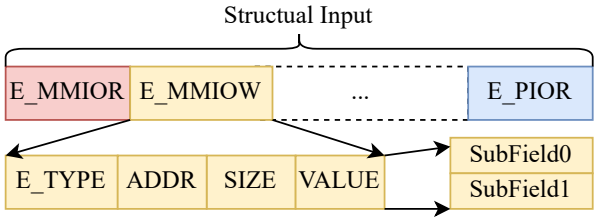


图 4.1 虚拟外设消息和结构化输入示例

```
1  typedef struct {
2    uint32_t command; uint32_t array_addr; } tx_t;
3  void action_command(physaddr addr) {
4    tx_t tx;
5    MacAddr macaddr;
6    TxConfig config;
7    dma_read(/*addr=*/addr, /*dst=*/&tx);
8    switch (tx.command & COMMAND/*=7*/) {
9    case CmdIASetup/*=1*/: dma_read(tx.array_addr, &macaddr); break;
10   case CmdConfigure/*=2*/: dma_read(tx.array_addr, &config); break;
```

图 4.2 当字段 array_addr 中的标志位不同时，字段 command 可以指向不同的缓冲区

4.2 挑战和观察

虚拟设备消息定义了消息结构和消息字段。图 4.1显示了一个有四个字段的 MMIO 写消息：一个类型字段 E_TYPE 和三个参数字段：ADDR，SIZE 和 VALUE。此外，VALUE 包含具有附加信息的格式化的子字段。多个消息组成了必须遵循特定顺序的消息序列。

利用虚拟外设消息进行模糊测试有三个关键好处。第一，有了明确的格式，模糊测试器就有了对输入的精确感知，从而可以支持细粒度的依赖关系。第二，虚拟外设消息允许模糊测试器通过操纵消息序列的顺序来建立虚拟外设的内部状态。第三，模糊测试器可以通过删除不必要的虚拟外设消息来找到有趣的消息序列，从而使 PoC 最小化。

之前的工作[49,54-55] 证明了基于覆盖率的模糊测试器适用于虚拟外设。然而，从虚拟外设消息的角度来看，有两个挑战被忽略了：消息内和消息间的依赖关系。

消息内依赖关系。一个虚拟外设消息通常包含多个字段。这些字段可能是相互依赖的。当虚拟外设通过 DMA 通道[55]与主内存大量交互时，通常会出现这种情况，DMA 通道可以处理大量数据块。例如，图 4.2显示了在行 7向 tx 的数据加载。然后，在行 8通过检查 command 的前三个比特来决定 array_addr 的类型。对这种依赖关系的认识减少了搜索空间（我们只突变了 command 的前三位），这就排除了模糊测试器的随机猜测，以满足复杂的约束（我们知道 array_addr 要么是 MacAddr 要么是 TxConfig）。

```
typedef GlobalState {
  uint32 internal_state; } GlobalState;
GlobalState gs;
void mmio_write_dword(
    physaddr addr, uint64_t val) {
  switch (addr) {
    case 0x0:
      gs->internal_state = val; break;
    case 0x4:
      if (!gs->internal_state) break;
      // do something and break
```

图 4.3 有序的信息序列 {addr: 0x0, val: rand()}, {addr: 0x4, val: rand()} 触发行 11

消息间依赖。虚拟外设消息可以被连锁，建立起复杂的虚拟外设状态。由于寄存器的范围很窄，与设备的交互往往需要多次交互。图 4.3展示了特定的消息序列如何触发行 11的代码。

4.3 核心算法之依赖感知的消息模型

依赖感知的消息模型同时解决了消息内和消息间的依赖关系。消息内的依赖性由一个轻量级的描述性语法（小节 4.3.1）来实现，而消息间的依赖性则由一组专门的消息突变器（小节 4.3.2）来实现。本节是系统设计（节 4.4）和系统实现（节 4.5）的基础。

4.3.1 消息内依赖注释

对于消息内依赖性，我们提出了一个新的轻量级描述性语法，在图 4.4中得到了体现。ViDeZZo 使用该语法来生成满足消息内依赖性的虚拟外设消息。由于注释是依赖于设备的，我们开发了一个注释推理引擎，从虚拟外设源代码中提取规范（小节 4.5.1）。

与 Syscall 描述语言（Syzlang）[68]不同，每个虚拟外设都建立在简单接口之上的临时协议（例如，在 表 2.2中只有四种类型的消息），这很容易被一个通用语法建模。就我们所知，我们是第一个将这种技术应用于虚拟外设领域的人。与 Nyx-Spec 不同，分析人员不需要复杂的硬件级规范知识。与不了解硬件规范的模糊测试器相比，如 V-Shuttle 或 MorPhuzz，我们的轻量级语法能够产生更高质量的种子。

我们的语法基于对五个类别的 18 个设备的分析。该语法依赖于一个小型的类型系统、API 和语句规则。类型系统和 API 涵盖了三个要求，即字段感知、位感知和上下文感知。而语句规则定义了如何开发注释。

```
// type system
FIELD_TYPE: RANDOM | CONSTANT | POINTER | FLAG

FIELDNAME: NAME
FIELD : FIELDNAME '#' SIZE
typedef uint8_t  BEGIN
typedef uint8_t  LENGTH
typedef uint32_t INITVALUE
FLAG_LEN_PAIR: BEGIN ':' LENGTH [ '@' INITVALUE ]

// APIs
STRUCTNAME: NAME
STRUCT_SET: '[' STRUCTNAME+ ']'
FIELD_INDEX : STRUCTNAME '.' FIELDNAME
FIELD_TYPE_PAIR: FIELD ':' FIELD_TYPE
FIELD_SET: '{' FIELD_TYPE_PAIR+ '}'
FLAG_SET: '{' FLAG_LEN_PAIR+ '}'
CONDIDATES: '[' uint32_t+ ']'
POINT_TO_SET: '[' FIELD_INDEX+ ']'
CONDITION: FIELD_INDEX '.' BEGIN
CONDITION_SET: '[' CONDITION+ ']'

def add_struct(
  name :-> STRUCTNAME, fields :-> FIELD_SET)
def add_flag(
  field :-> FIELD_INDEX, flags :-> FLAG_SET)
def add_constant
  field :-> FIELD_INDEX,
  condidates :-> CONDIDATES)

def add_head(structs :-> STRUCT_SET)
def add_point_to(
  field :-> FIELD_INDEX,
  point_to :-> POINT_TO_SET,
  condition :-> CONDITION_SET,
  ALIGNMENT :-> uint8_t)

def add_point_to_linked_list(
  head :-> FIELD_INDEX, tail :-> FIELD_INDEX,
  point_to :-> POINT_TO_SET, links :-> FIELD_SET,
  condition :-> CONDITION_SET,
  ALIGNMENT :-> uint8_t)

// statements and programming model
MODELNAME: NAME
model: 'Model(' MODELNAME ',' MODELID ')'
api_add_field:
  add_flag | add_constant | add_point_to
statement: add_struct add_field+
statements: model statement+ add_head
```

图 4.4 描述消息间依赖的语法

类型系统。该语法有一个类型系统来反映虚拟外设消息的字段感知和比特感知，这由我们的研究第一次系统地引入。

- 字段感知。如果一个虚拟外设消息为其字段和子字段定义了边界和类型，那么它

就是字段感知的。具体来说，子字段是数据字段的子组件，如 图 4.1所示。子字段应该被认为是独立的，可以包含数据或指针。数据字段的值可以是随机的，也可以是常数；而指针字段则是把嵌套的对象连起来[55]。例如，在图 4.2中，command 和 array_addr 都是四个字节。前者是一个数据字段，后者是一个指针字段。我们的语法允许我们明确地注释这些信息。

- 比特感知。如果一个虚拟外设消息的字段在比特粒度上定义了约束，那么它就是比特感知的。例如，数据字段的一些位被用作标志，如行 8中图 4.2所示。同样地，指针经常在一些，例如较低的位上包括额外的信息作为标记。

这个语法（从行 2开始）首先定义了四个基本的字段类型，每个字段都有一个正交的符号，也就是说，RANDOM 代表一个具有随机值的变量，CONSTANT 代表一个常数，POINTER 代表一个指针，FLAG 代表一个具有标志位的变量。由于正交性，一个带有标记的指针可以是 POINTER | FLAG。然后，该语法定义了一个 FIELD，由其字段名和字段大小决定。比如说，FIELD command 是 command#0x4。接下来，它定义了如何表达标志位，即：FLAG_LEN_PAIR。例如，command 的前三个位是 0: 3@7，那么这三个位的初始值是 7；一般来说，0: 3 意味着这些位的初始值是随机的。

APIs 和上下文感知。我们的语法提供了注释消息内部依赖关系的 API。add_struct() 定义了虚拟外设消息中的每个字段，add_flag() 定义了标志位，add_constant() 定义了常量字段的候选值集。此外，我们可以通过 add_head() 定义头部（根部）对象和 add_point_to() 定义后代，来定义树状对象（每个节点都包含一个指向下一个节点的指针[55]）。

除了上述规则外，我们还确定了三种新型的上下文感知的依赖关系。具体来说，虚拟外设消息可能是上下文感知，也就是说，虚拟外设中的字段有特定的依赖关系。我们确定了三种上下文感知情况，如下所述。

- 头尾指针上下文。指针可以指向一个单一的对象或者一个对象的链表。链表可能需要一个额外的指针字段来指示链表的尾部（图 4.5），这是合理的，因为一些虚拟外设可以在一个消息中处理一连串（或环）的命令。我们通过 add_point_to_linked_list() 来支持，它与 add_point_to() 相似，但明确定义了一个链表。
- 标志位/标签指针上下文。字段中的标志位可能会影响模糊测试的执行路径，相应的指针有不同的类型。数据和指针字段都可以有标志位。这种依赖关系从一个位的数据字段开始，以一个多类型的指针字段结束。图 4.2显示了一个例子。为了支持这种

```
typedef struct {
  uint32_t head; uint32_t tail; } ed_t;
typedef struct {
  uint32_t next; } td_t;
void handle_end_descriptor(physaddr head)
  ed_t ed;
  dma_read(/*addr=*/head, /*dst=*/&ed)
  while ((ed.head & 0xffffff00) != ed.tail) {
    td_t td;
    phsyaddr addr = ed->head & 0xffffff00;
    if (ed.head & 0x1) /* be invalid and return */
    dma_read(/*addr=*/addr, /*dst=*/&td);
    ed->head |= td.next & 0xfffffff00;
//---------
vd1 = Model('ed', 1)
vd1.add_head(['ed_t'])
vd1.add_struct('ed_t', {'head#0x4': POINTER|FLAG, 'tail#0x4': 'POINTER'})
vd1.add_flag('ed_t.head', {0: 1@0})
vd1.add_struct('td_t', {'next#0x4': 'POINTER'})
vd1.add_linked_list('ed_t.head', 'ed_t.tail', ['td_t'], ['next'], alignment=8)
vd1.add_head(['ed_t'])
```

图 4.5 头尾指针上下文示例展示了指针字段的协作方式：指针字段 head 和 tail 指向 td_t 的单链表的头和尾；监控每个 dma_read()[49,55]的模糊测试器因不知道跨 dma_read() 的依赖关系，无法处理该上下文，如果不处理，因随机的 head 和 tail 不太可能相等，行 8中的 while 循环不能终止

依赖性，我们给 add_point_to() 添加了一个新参数 condition :-> CONDITION_SET。这个参数显示了哪些位将决定相应指针字段的类型。

- 长度缓冲区上下文。一个随机数据字段可能有特定的语义，以表明一个缓冲区有多长。只有当虚拟外设像校验算法一样检查缓冲区的长度时，才需要这种依赖性。这种依赖性需要额外的本地约束分析，例如，符号执行，以及神谕来告诉分析的位置。图 4.6显示了一个例子。为了支持这种依赖性，我们将数据字段的类型从 RANDOM 调整为 CONSTANT，并将缓冲区的长度放到候选值列表中。

只要保持向后兼容，我们的语法可以被扩展。图 4.7列出了额外的消息内依赖关系。

语句和编程模型。除了类型系统和 API，我们还引入了一个编程模型（从行 45），供分析人员开发消息内注释。每个虚拟外设都被编码为一个或多个模型（Model），包含了消息内的依赖关系。分析师可以通过包括头部对象（add_head()）、结构（add_struct()）和字段（add_field()）来扩展一个模型。通过这种方式，一个缓冲区树被建立起来，以支持字段感知、位感知和上下文感知。图 4.8显示了图 4.2的消息内依赖的注释。

完全自动化的注释提取是具有挑战性的。我们的半自动化注释提取大大减少了人力的付出，在支持新的虚拟外设上，这比人工检查虚拟外设规格更具扩展性。我们在小节 4.5.1中总结了实现情况，在小节 4.6.1中总结了注释提取过程中的不确定性。

```
typedef {
  uint64_t addr1; uint32_t len; } bpl_t;
void handle_hda(physaddr addr0)
  bpl_t bpl;
  dma_read(/*addr=*/addr0, /*dst=*/&bpl);
  int n_copied = custom_memcpy(/*src=*/bpl.addr1, /*dst=*/buf);
  if (bpl.len == n_copied) {
    // do something
//----------
vd2 = Model('bpl', 2)
vd2.add_head('bpl_t')
vd2.add_struct('bpl_t', {'addr1#0x8': 'POINTER', 'len#0x4': 'CONSTANT'})
vd2.add_strcut('bpl_buf', {'buf#0x1000': 'RANDOM'})
vd2.add_point_to('bpl_t.addr1', ['bpl_buf'])
vd2.add_constant('bpl_buf.len', [0x1000])
```

图 4.6 长度缓冲区上下文显示了一个数据字段如何与一个指针字段协作：如果模糊测试器不控制 len 的值，那么在行 7的分支就不可能被执行

表 4.1 多级突变器和它们的描述汇总：ML 指 Message-level，SL 指 Sequence-level，GL 指 Group-level

ID	Lvl	Mutators	Description
1	ML	ChangeValue	Mutate the value of a message
2		ChangeAddr	Mutate the address of a message
3		ChangeSize	Mutate the size if not fixed
4		EraseMessage	Randomly erase a message
5		InsertMessage	Insert one new message
6		InsertRptdMessage	Insert multple new messages
7	SL	ShuffleMessages	Shuffle a sequence of messages
8		CopyPartOfSequence	Copy messages from a sequence to another
9		CrossOverSequence	Exchange messages between two sequences
10		EraseSequence	Randomly erase part of a sequence
11		InsertSequence	Insert a sequence randomly
12		ShuffleSequence	Shuffle all messages in a sequence
13	GL	GroupMessage	Group messages for future re-use

4.3.2 消息间突变器

为了解决消息间依赖，我们提出了三类新的消息突变器，通过编排消息和消息序列（表 4.1）来推断消息间的依赖关系。我们的自动方法克服了繁琐的手工追踪和建模[59]。

我们的突变器提供了一致性（消息级）、多样性（序列级）和语义（组级）。据我们所知，我们是第一个将这种技术应用于虚拟外设领域的人。我们的突变器设计遵循这一直觉：一旦模糊测试引擎发现一个有趣的种子，它就会把这个种子保留在语料库中，并在下一个模糊循环中持续突变它。换句话说，消息间的依赖关系并没有被具体定义，而是逐渐被学习。我们在下文中详细介绍了突变器。

- 消息级突变器。我们定义了六个消息级的消息突变器，这些突变器修改了消息的

```
1  static void xhci_doorbell_write(
2      void *ptr, hwaddr reg, uint64_t val, unsigned size) {
3    reg >>= 2;
4    if (reg == 0) {
5      if (val == 0) {
6        xhci_process_commands(xhci);
7    } else {
8      epid = val & 0xff;
9      streamid = (val >> 16) & 0xffff;
10     xhci_kick_ep(xhci, reg, epid, streamid);
11 }
12 //----------
13 vd3 = Model('xhci_doorbell_write_0', 3)
14 vd3.add_head('mmio_write')
15 vd3.add_struct('mmio_write', {
16   'addr#0x8': 'CONSTANT', 'len#0x4': 'CONSTANT', 'valu#0x8': 'CONSTNAT'})
17 vd3.add_constant('mmio_write.addr', 0x0)
18 vd3.add_constant('mmio_write.len', 0x4)
19 vd3.add_constant('mmio_write.valu', 0x0)
20
21 vd4 = Model('xhci_doorbell_write_!0', 4)
22 vd4.add_head('mmio_write')
23 vd4.add_struct('mmio_write', {
24   'addr#0x8': 'CONSTANT', 'len#0x4': 'CONSTANT', 'valu#0x8': 'FLAG'})
25 vd4.add_constant('mmio_write.addr', [i for i in range(4, 0x20)])
26 vd3.add_constant('mmio_write.len', 0x4)
27 vd4.add_flag('mmio_write.valu', {0: 8, 8: 16, 16: 32, 32: 64@0})
```

图 4.7 MMIO 访问里面的消息内依赖示例：首先，行 6要求作为 MMIO 写消息的字段的 reg 和 val 都要等于 0，否则，xhci_process_commands() 不能执行；其次，行 10强调了 val 中的子字段

内容，保证了原始消息和突变后的消息之间的一致性。例如，MMIO 地址的改变可以产生对虚拟外设的连续访问。具体来说，我们定义了五个主要的突变器（ID 1–5）和一个拓展的突变器（ID 6）。突变器 1–3 修改消息中的每个参数，以保持消息的平衡性。突变器 4–5 随机删除和插入一条信息以调整输入的长度。突变器（ID 6）增加几个重复的信息。根据我们的观察，这个突变器会增加或减少虚拟外设中的变量，让模糊测试器绕过边界检查。

- 序列级突变器。我们定义了六个序列级的消息突变器，在一个序列中增加或删除消息。与通过消息级突变器进行的局部修改一起，这导致跨消息序列的剧烈变化的概率更高，以绕过较难绕过的检查[97]。具体来说，突变器 7–9 更新序列内的消息，而突变器 10–12 处理序列。
- 成组突变器。我们定义了新的“成组突变器”，其目的是将依赖性的消息分组为一个所谓的 GROUP_MESSAGE。被分组的消息在后续的突变中保持不变。这些突变器是建立在“触发行动协议”之上的，它依赖于“反馈”（触发）和“一个处理程序”（行动）来决定是否对消息进行分组。小节 4.4.3详细介绍了两类成组突变器。

```
vd0 = Model('tx', 0)
vd0.add_head(['tx_t'])
vd0.add_struct('tx_t', { 'command#0x4': 'FLAG', 'array_addr#0x4': 'POINTER'})
vd0.add_flag('tx_t.command', {0: 3})
vd0.add_point_to('tx_t.array_addr',
  [None, 'macaddr', 'config', None, None, None, None, None],
//if 0       1          2       3     4     5     6     7
  condition=['tx_t.command.0'])
//          == tx_t.command.0
```

图 4.8 消息内依赖注释示意图：tx_t.command.0 **的值决定了** tx_t.array_addr **的类型**

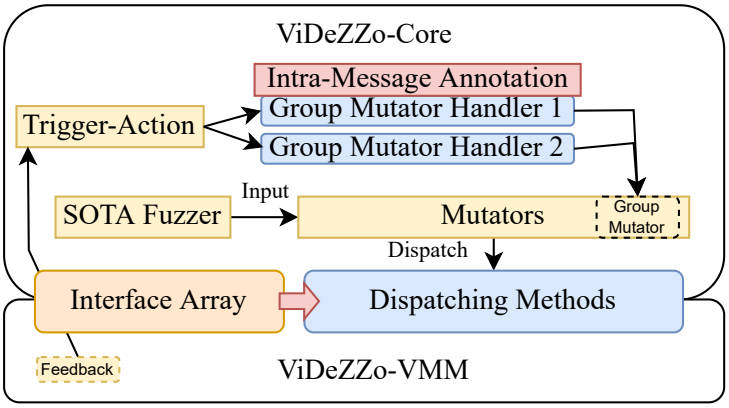


图 4.9 ViDeZZo 的系统设计示意图

4.4　依赖感知的虚拟外设模糊测试框架的系统设计

ViDeZZo（图 4.9）由两个组件组成：ViDeZZo-Core，它与 VMM 无关，提供由虚拟外设消息组成的输入；ViDeZZo-VMM，它是 VMM 专用的，与虚拟外设互动。具体来说，首先，ViDeZZo 从队列中挑选一个输入，即一个消息序列，并将其传递给消息和序列级的突变器（小节 4.4.1）。接下来，为了分发消息，从共享测试接口数组中获取真实物理地址，并调用相应的分发方法（小节 4.4.2）。然后，ViDeZZo 使用来自虚拟外设的反馈来行使触发行动协议。按照这个协议，成组突变器将消息分组（小节 4.4.3），例如，将从消息内标记生成的消息（小节 4.4.4）分组。

图 4.10显示了模糊测试循环的工作流程。在开始时，模糊测试器从语料库中选择一个种子。如果语料库是空的，那么第一个种子就是空的。在这个例子中，种子由四条信息组成，消息 1–4。为了简单起见，我们隐藏了它们的内部格式。然后，种子被突变为一个测试案例：消息 1 被保留；标记在消息 2 的深灰色区域的参数被更新为另一个值（突变器 1，2 或 3）；消息 3 被删除（突变器 4）；消息 4 被保留；另一个带有不同参数的消息 1 被添加（突变器 5）。在种子突变之后，模糊测试器依次将消息分发给目标虚拟

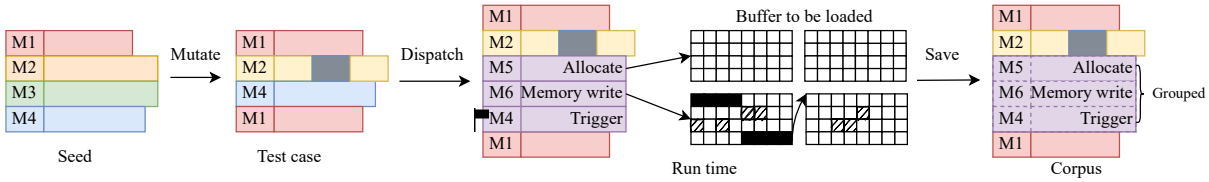


图 4.10 在灰盒模糊测试下的 ViDeZZo 的工作流：M 指 Message

外设。在这个例子中，消息 1 和消息 2 首先被分发。接下来，一个“触发”通知模糊测试器，消息 4 即将从主存储器加载数据。紧接着一个“行动”，模糊测试器生成并注入消息 5 和 6，在消息 4 加载数据之前设置主存储器。请注意，注入的消息和触发消息在下一个模糊循环中被锁定并作为一个组进行变异（突变器 13）。最后，模糊测试器分发了最后一个消息 1，并完成了测试用例的执行。最后，如果测试用例发现了新的代码覆盖，模糊测试器将变异的种子保存到语料库中，并重复该循环。

4.4.1　输入解析和第一次变异

ViDeZZo-Core 通过序列化和反序列化将字节数组种子解析为消息序列。接下来，我们的消息级和序列级突变器会生成一个新的输入。小节 4.5.2介绍了该实现。

4.4.2　接口和分发方法

在突变（消息和序列级突变器）之后，模糊测试器通过调用相应的 VMM 特定的分发方法来分发每个虚拟外设消息。具体来说，ViDeZZo-Core 定义了一组分发接口，在 ViDeZZo-VMM 中被实例化。这种基于接口的抽象允许 ViDeZZo 读取和写入虚拟外设的内部状态，便于 ViDeZZo 扩展到更多的虚拟外设类别、架构和虚拟机管理程序。

这种机制使用测试接口阵列来定义允许哪些消息以及如何分发这些消息。例如，如果一个虚拟外设的接口描述了一个从 0xe0001000 到 0xe0001200 的四字节对齐的 MMIO 区域，则允许客户在该范围内发出 MMIO 读写消息。注意，请求必须是四字节对齐的。几个测试接口组成了测试接口阵列，在 ViDeZZo-Core 中定义，在 ViDeZZo-VMM 中实例化。该阵列包括动态接口和预定义接口，如下所述。

- 动态接口。对于每个模糊测试实例，ViDeZZo 只会对一个虚拟外设进行模糊测试，从而提高虚拟外设消息的利用率。对于目标虚拟外设，ViDeZZo 找到并提取其内存区域对象，这些对象定义了 PIO 或 MMIO 内存空间的起始地址、大小和对齐

方式等信息，这是可行的，因为每个 VMM 在注册虚拟外设时都设置了一个定义明确的结构。通过 VMM 特定的 API 获得这些信息需要预定义的接口签名。例如，EHCI 的一个内存区域与作为签名的字符串 “capabilities” 相关。

- 预定义接口。ViDeZZo 定义了几个不能自动识别的接口，如内存分配、读写和释放接口。具体来说，除了 PIO 和 MMIO 相关的接口，我们对内存和时钟相关信息的接口进行编码。这些固定接口的数量是有限的，一旦定义，ViDeZZo 就会在不同的目标上使用它们。

我们通过一个额外的 interface_id 字段将每个虚拟外设消息与一个给定的接口联系起来。当分发一个消息时，分发器通过 interface_id 从测试接口数组中查找合适的接口，调整目标地址，校准数据，然后调用相应的分发方法。

4.4.3 成组突变器

ViDeZZo 提出了两个成组突变器：Load Miss 突变器和 Record Start-End 突变器。成组突变器是通过新提出的触发行动协议来实例化的。该协议有一个定义明确的触发（反馈）和一个连接到触发器的行动（处理程序）。该协议像一个中断服务程序，使上下文在 ViDeZZo-VMM 和 ViDeZZo-Core 之间切换。我们区分了三个阶段。首先，当反馈被击中时，模糊测试器停止当前的执行，转到处理程序。其次，处理程序将相关的消息分组为一个 GROUP_MESSAGE。最后，处理程序返回，模糊测试器恢复执行。

- Load Miss 突变器。一些虚拟外设期待一组相互链接的消息[55]，我们引入了 Load Miss 突变器来处理这种情况。图 4.10显示了一个例子：在消息 4 发出后，一个 pci_dma_read() 被调用了。然而，ViDeZZo 必须注入消息 5 和 6 以满足消息内的依赖性，否则，虚拟外设将读取无意义的数据。对于 “反馈”，当虚拟外设试图从客户虚拟机内存加载数据时，我们拦截 load 指令，称这种反馈为 Load Miss。对于 “行动”，突变器实现了一个处理程序，名为 Load Miss 处理程序，需要 load 的目标地址。在处理程序中，ViDeZZo 生成了一组与原始信息相联系的信息（小节 4.4.4）。
- Record Start-End 突变器。一些虚拟外设需要按特定顺序排列消息，如图 4.3中的行 11所示。为了解决这个要求，我们引入了一个 Record Start-End 突变器，记录与共享变量有关的有序消息，例如 gs->internal_state。我们定义了 “反馈” 的两个点：start 和 end。当共享变量被写入时触发了 start 反馈，而 end 反馈是在共享变量被使用

算法 2: Annotation-to-Message Construction

```
   Input: Intra-Message Annotation D
   Input: Message memalloc, memread, memwrite, memfree
   Result: Messages M = {m_1, m_2, …, m_n}
1  Function Alloc(Size):
2      M.append (memalloc (Size));
3  Function Fillup(Object):
4      foreach Field ← Object.Fields do
5          Metadata ← Field.Metadata
6          switch Field.Type do
7              case FLAG do
8                  M.append(memwrite(gen_flag(Matadata))); break;
9              case CONSTANT do
10                 M.append(memwrite(Field.Value)); break;
11             case RANDOM do
12                 M.append(memwrite(rand())); break;
13             case POINTER do
14                 M.append(memwrite(gen_pointer(Matadata))); break;
15 Function Free(Addres):
16     M.append(memfree(Address));
17 Head ← D.Head
18 HeadAddress ← Alloc (sizeof (Head));
19 Fillup (Head);
20 Free (HeadAddress);
```

时。在 start 时，成组突变器将记录所有后续信息，直到 end 被触发。在“行动”中，ViDeZZo 将记录的消息分组并保持完整。

注意，成组突变器与其他两级突变器一起工作，自动学习消息间的依赖关系。目前，我们支持两个成组突变器。尽管如此，开发者可以在我们的触发行动协议之上实现其他反馈和处理程序。例如，将一组固定的虚拟外设消息分组，以重置一个虚拟外设。

4.4.4　消息内依赖标记到消息

算法 2展示了如何利用消息内注释来生成虚拟外设消息，以支持消息内的依赖性。为了更好地理解，我们定义了三个高层次的概念性消息：Alloc、Fillup 和 Free。它们中的每一个都由表 2.2中定义的一个或几个基本消息组成。

构造过程从头部对象开始（第 17–20 行），它依次调用 Alloc(), Fillup(), 和 Free()。Alloc()（第 1–2 行）将虚拟外设消息 memalloc（例如，在图 4.10中的信息 5）添加到消息集 M。此外，Alloc() 在客户内存中分配了一个缓冲区。Fillup()（第 3–14 行）添加了

memwrite（例如，图 4.10中的消息 6），向头部对象中的每个字段写入一个特定或随机的值。如果一个字段是一个标志或一个指针，Fillup() 会根据它的注释来设置它的值。例如，一个标志，其标志长度对为 $0:16, 16:16$，其值为：(rand() && 0xff) |((rand() && 0xff) $\ll$ 16)。如果一个字段是一个指针，那么 Fillup() 就会获得它的指向对象，检查是否存在需要引用的条件字段，并递归地调用 Alloc() 和 Fillup() 来生成一个普通对象或一个链表。在 Alloc() 和 Fillup() 之后，Free()（第 15-16 行）释放了头部对象。注意，为了简化算法，我们假设不再使用的对象被释放，以避免客户虚拟机内存的耗尽。

4.5 依赖感知的虚拟外设模糊测试框架的系统实现

我们用 364 行 CodeQL 脚本实现了自动化的注释提取，用 812 行 Python 实现了消息内标记，用 1855 行 C 和 518 行 Python 实现了 ViDeZZo-core，用 1503 行 C 实现了 ViDeZZo-QEMU，最后，用 1165 行 C 实现了 ViDeZZo-VirtualBox 。ViDeZZo 支持最新的 QEMU 和 VirtualBox。我们在 https://github.com/HexHive/ViDeZZo发布我们的工具。在下文中，我们将讨论具体的 ViDeZZo 模块。

4.5.1 半自动化的消息内依赖注释提取

为了帮助分析，我们引入了一个引擎用于提取消息内的依赖关系。我们的工具生成了一个独立的文本文件，其中有对字段和比特感知的基线注释。分析师可以在以后完善该基线。我们在 CodeQL 的基础上开发了这个引擎，它整合了数据流和污点分析。

- 字段感知。从 DMA 读取访问开始，例如，pci_dma_read()，我们的工具提取目标缓冲区的结构定义。对于结构中的每个字段，该工具通过污点分析检查该字段是否流入 DMA 访问代码，如 pci_dma_[read|write]()，从而检查它是否是一个指针。如果该字段是一个指针，该工具会进一步获取其可用的类型。在少数情况下，虚拟外设使用字节数组，并且不对目标缓冲区使用任何转型，这就禁止了显式结构定义。这种结构重构需要额外的模型来将真实的结构定义映射到数组上。我们开发了一个 Python 脚本来将小的字节数组（根据我们的经验，小于或等于 32 字节）分割成四个字节的小块，并将每个小块标记为数据。
- 比特感知从上面得到的结构定义开始，我们的工具以结构字段的访问为操作数，遍

表 4.2 虚拟外设消息的组成汇总

Message Type	TYPE	INTERFACE_ID	ADDR	SIZE	VALUE
[MMIO/PIO]_READ	1	1	8	4	—
[MMIO/PIO]_WRITE	1	1	8	4	8
MEM_[READ/WRITE]	1	1	8	4	SIZE
MEM_[ALLOC/FREE]	1	1	—	—	8
CLOCK_STEP	1	1	—	—	8

历所有二进制操作。该工具对二进制操作进行解析，并确定哪些位应该被考虑。

4.5.2 ViDeZZo-Core

ViDeZZo-Core 与 VMM 无关，负责对输入信息进行解码和编码，并对虚拟外设消息进行变异。我们的 ViDeZZo-Core 基于 libFuzzer 实现。

输入解析器。在表 4.2中，我们列出了表 2.2中定义的所有消息格式。每个消息都至少包含两个字节，定义了消息类型和它的接口。通常，addr 是八个字节，size 是四个字节。至于 value，大多数是八个字节，但对于 MEM_[READ/WRITE]，value 的大小取决于专用的 size 字段。这样实现的原因是为了预先分配一个大的缓冲区，减少内存分配的开销。最后，为了对输入进行解码和编码，我们通过 next_##size##_bytes() 胶水函数实现了一个序列化器和一个反序列化器。

消息级和序列级突变器。我们通过覆盖 libFuzzer 中的 LLVMFuzzerCustomMuatator() 实现我们的突变器。首先，我们将二进制输入反序列化为消息对象并直接对其进行操作。在突变之后，我们将消息序列化为一个新的二进制输入。由于我们对每个输入只进行一次反序列化，所以突变操作很轻量。

成组突变器。触发动作协议有两部分。第一部分是目标虚拟外设中的触发插装，第二部分是在 ViDeZZo-Core 中实现相应的处理程序。对于第一部分，我们建立了一个基于 Clang 的插装 pass，利用目标虚拟外设中安全分析师的注释进行插装；对于第二部分，分析员应该开发他们自己的处理程序。

图 4.11显示了 Load Miss 突变器的实现。通过钩住 QEMU 中的 pci_dma_read() 和 VirtualBox 中的 PDMDevHlpPCIPhysRead() 来检测 Load Miss。目标缓冲区的地址被传递给 Load Miss 处理程序。这个处理程序是与 VMM 无关的，在 ViDeZZo-Core 中实现。最后，我们恢复对虚拟外设的控制流。

```
static int __wrap_pci_dma_read(
    uint32_t addr, void *buf, size_t size) {
  /*handler*/LoadMissHandler(addr);
  return REAL(pci_dma_read)(addr, buf, size);
}

void ehci_state_fetchqh(EHCIState *ehci) { EHCIqh qh;
  /*feedback*/WARP(pci_dma_read)( addr, &qh, sizeof(EHCIqh));
```

图 4.11 Load Miss 突变器的插装：我们使用 pci_dma_read() 来与论文中的行话保持一致，而我们在实现中使用 get_dword()

```
void mmio_write_dword(physaddr addr, uint64_t val) {
  switch (addr) {
    case 0x0:
      gs->internal_state = val; break;
      /*feedback*/Record(0, /*mode=*/"start");
    case 0x4:
      if (!gs->internal_state) break;
      /*feedback*/Record(0, /*mode=*/"end");
      // do something and break
```

图 4.12 Record Start-End 突变器的插装示意图

图 4.12显示了 Record Start-End 突变器的实现。我们引入了一个名为 Record(int id, char *mode) 的 API，其第一个参数是一个唯一的标识符，第二个参数表示是否开始或结束记录。Record() 函数追踪所有的信息，并在记录结束时将其分组。

在我们的原型中，所有的反馈都由自动生成的虚拟外设消息触发。另一种策略则需要仔细地手工编排消息。然而，我们认为这种方法是不可行的，因为一个虚拟外设可能需要一长串满足复杂消息间依赖关系的信息。相反，我们的触发行动协议避免了这一挑战，并简化了设计和实现。此外，基于我们的协议，保存的崩溃测试案例是完整的，不需要再生[55]或消息重新排序[49]。

从消息内标记到消息。根据算法 2，我们用 Python 实现了一个语法解释器，它将描述自动翻译成 C 代码的消息生成器。然后，这些生成器与 ViDeZZo-Core 一起被编译。

持久模糊测试。重新启动虚拟机是一个耗时的操作，限制了模糊测试器的性能。为了克服这一限制，我们实现了一个进程内的持久性模糊测试器。然而，持久性模糊测试器在执行测试案例时，会积累以前的状态。这种积累可能会导致不可复制的崩溃，因为一个错误可能来自之前的多个测试案例。因此，我们修改了 libFuzzer，以选择性地记录所有中间测试案例（类似于 Syzkaller）。然后，我们用 Picire[96]实现差分调试方法，推断出负责该崩溃的测试用例，用剩下的测试用例推断出负责同一崩溃的虚拟外设消息。差

分调试对连续的种子和消息进行二进制搜索，建立最小崩溃种子，减少了分析消息的工作量。根据我们的经验，最小化的种子通常小于 80 条消息（见表 4.9的最后一列）。

4.5.3 ViDeZZo-VMM

ViDeZZo-VMM 是针对 VMM 的，但是，它遵循一个固定的模板。ViDeZZo-VMM 有四个步骤：(1) 注册目标虚拟外设，(2) 初始化虚拟机，(3) 用真实物理地址设置共享接口阵列，(4) 实现 VMM 特有的分发方法。总结的模板突出了扩展到新的 VMM 和相关虚拟外设的必要知识，避免了开发者迷失在虚拟机管理程序的巨大代码空间中。目前，ViDeZZo 支持 QEMU 和 VirtualBox。在下文中，我们将详细介绍这些步骤。

目标虚拟外设注册。为了注册一个目标虚拟外设，ViDeZZo-VMM 提供了相应的规范：架构、启动命令行和模糊测试接口的签名。最重要的是，启动命令行是针对目标的，而测试接口的签名则取决于虚拟机管理程序。对于规范格式，我们调整了 QEMU-Fuzzer[98]中的现有实现，增加了一个新字段 mrnames 作为测试接口的签名。我们为每个管理程序添加了涵盖不同类别和架构的虚拟外设，如下一节实验所示。

VMM 初始化和接口识别。为了初始化一个 VMM，我们将目标设备的规范中的启动命令行传递给 VMM 的 main() 函数。然后，为了识别虚拟外设中的非预定义测试接口，我们扫描所有注册的 PIO 和 MMIO 内存区域对象，并选择与签名匹配的接口。最后，我们提取元数据（例如，所选 PIO 或 MMIO 内存区域的物理地址），并将元数据填入共享接口阵列。我们重用 QEMUFuzzer[98]中的代码，用于 ViDeZZo-QEMU，并为 ViDeZZo-VirtualBox 实现了类似功能。

VMM 特定的消息分发。我们为 ViDeZZo-QEMU 使用了 QTest APIs。QTest 是一个基于 QEMU 消息协议（QMP）的进程内测试框架。QTest 可以直接访问客户内存，并可扩展到多个虚拟外设和架构。我们还为 ViDeZZo-VirtualBox 实现了一个类似的功能。

4.6 实验验证

本节介绍了对 ViDeZZo 的评价，我们设计这个评价是为了回答以下四个研究问题。

RQ1: 与其他虚拟外设模糊测试相比，ViDeZZo 的总体效率如何？

RQ2: 与依赖关系无关的模糊测试器相比，ViDeZZo 的覆盖率和开销有什么不同？

RQ3: 与现有的工具相比，使用 ViDeZZo 来发现现有的安全缺陷有什么优势？

RQ4: ViDeZZo 在发现新的安全缺陷上表现如何？

模糊测试器和虚拟机管理程序设置。对于 **RQ1–RQ3**，我们移植了 (i)Nyx-Legacy 和 (ii)Nyx-Legacy。对于 **RQ1–RQ3**，我们将（i）Nyx（包括 Nyx-Legacy 和 Nyx-Spec）、QEMUFuzzer（MorPhuzz 的“工业”版本）和 ViDeZZo 移植到 QEMU 5.1.0;（ii）ViDeZZo 移植到 VirtualBox 6.1.14。目前，部分 Nyx、Morphuzz 和 V-Shuttle（也被称为 V-Shuttle-S）是开源的。Nyx 和 Morphuzz 支持最近的 QEMU，但它们不支持 VirtualBox。V-Shuttle 支持 QEMU 5.1.0 和 VirtualBox 6.1.14。对于 **RQ4**，我们将 ViDeZZo 升级到最新的 QEMU 和 VirtualBox。我们用 Clang 编译所有的虚拟机管理程序；我们不使用任何初始种子。

虚拟外设模糊测试设置。对于 **RQ1 和 RQ2**，我们选择 28 个虚拟外设，覆盖五个虚拟外设类别（USB，net，display，audio 和 storage）、四个架构（i386，x86_64，AArch32 和 AArch64）和两个虚拟机管理程序（QEMU 和 VirtualBox）。

覆盖率和漏洞检测器设置。对于 **RQ1 和 RQ2**，我们依靠 Clang 源代码覆盖率分析[99]。除了 Nyx，我们使用一个信号处理程序来转储实时覆盖率。具体来说，我们每秒钟转储一次覆盖率，以监测前 10 分钟的快速覆盖率变化，然后每 10 分钟转储一次以节省空间。对于 Nyx，我们重新执行所有的种子来收集离线覆盖。为了消除早期崩溃的干扰，我们禁用了所有的消毒剂，删除了所有的断言和 abort()，并修补了在目标中发现的任何错误。请注意，我们对所有的模糊测试器使用相同的补丁。当管理程序崩溃或超时时，我们进一步转储覆盖率以保证可靠的覆盖率收集。对于 **RQ3 和 RQ4**，我们禁用覆盖率分析，启用 ASan 和 UBSan，并保留所有断言和 abort() 以捕获更多的漏洞。

服务器资源。对于 **RQ1–4**，我们在六台禁用超线程的服务器上进行所有实验，每台服务器有 16 个 Intel Xeon Gold 5218 CPU（2.30GHz）内核，64GB 内存和 Ubuntu 20.04。我们在一个核心上从头开始对每个虚拟外设进行模糊测试，持续 24 小时，我们将每个实验重复 10 次以获得统计学意义[100]。

表 4.3 半自动消息内注释推断的统计结果：我们列出了支持的通用结构（我们使用不同的后缀来区分，例如，S 代表结构，A 代表数组，32 代表 uint32_t），标志字段、指针字段、结构的所有字段的数量（验证结果与自动化提取结果），还有虚拟外设的上下文感知依赖（I 代表头尾指针上下文，II 代表标志/标签指针上下文，III 代表长度缓冲上下文）

Device	Generic Struct	# of Flags	# of Pointers	# of Fields	# of Context -Awareness
AC97	AC97_BD.8.A	2/2	1/1	2/2	—
AC97	AC97_TMPBUF.8.A	—	—	1/1	—
CS4231a	CS4231A_BUF0.8.A	—	—	1/0	—
ES1370	ES1370_TMPBUF.8.A	—	—	1/1	—
Intel-HDA	INTEL_HDA_BUF0.8.A	1/0	1/0	3/4	III
Intel-HDA	INTEL_HDA_VERB.32	1/1	—	1/1	—
SB16	SB16_BUF0.8.A	—	—	1/0	—
AHCI	AHCI_CMFIS.8.A	2/0	—	37/32	—
AHCI	AHCI_SG.S	—	1/1	3/3	—
AHCI	AHCI_RESFIS.8.A	—	—	1/1	—
AHCI	AHCI_LST.8.A	—	—	1/1	—
FDC	FLOPPY_BUF.8.A	—	—	1/0	—
MEGASAS	MEGASAS_REPLY_QUEUE_TAIL.32	—	—	1/1	—
MEGASAS	MEGASAS_REPLY_QUEUE_HEAD.32	—	—	1/1	—
MEGASAS	MEGASAS_MFI_FRAME_HEADER_SENSE.64	—	—	1/2	—
MEGASAS	MEGASAS_MFI_INIT_QINFO.S	1/1	—	5/5	—
MEGASAS	MEGASAS_MFI_FRAME_INIT.S	1/1	2/1	15/15	II
MEGASAS	MEGASAS_MFI_FRAME_DCMD.S	1/1	—	17/17	II
MEGASAS	MEGASAS_MFI_FRAME_ABORT.S	1/1	—	15/17	II
MEGASAS	MEGASAS_MFI_FRAME_SCSI.S	1/1	—	12/12	II
MEGASAS	MEGASAS_MFI_FRAME_IO.S	1/1	1/1	17/17	II
SDHCI	SDHCI_FIFO_BUFFER0.8.A	—	—	1/1	—
SDHCI	SDHCI_FIFO_BUFFER1.8.A	—	—	1/1	—
SDHCI	SDHCI_ADMA2.64	0/1	1/1	4/2	—
SDHCI	SDHCI_ADMA1.32	1/1	1/1	1/1	—
SDHCI	SDHCI_ADMA2_64.64	—	1/1	4/2	—
E1000	E1000_RX_DESC.S	2/1	1/1	6/6	—
E1000	E1000_TX_DESC0.S	2/2	1/1	3/3	—
E1000E	E1000_TX_DESC0.S	2/2	1/1	3/3	II
E1000E	E1000_CONTEXT_DESC.S	4/4	—	4/0	II
E1000E	E1000E_READ_RX_DESC.8.A	—	1/0	4/4	—

表 4.3 续上页

EEPRO100	MAC_ADDR0.8.A	0/1	—	6/6	II
EEPRO100	CONFIGURATION.8.A	22/2	—	22/22	II
EEPRO100	TX_BUFFER_ADDRESS.32	—	1/1	1/1	II
EEPRO100	TX_BUFFER_SIZE.16	—	—	1/1	II
EEPRO100	TX_BUFFER_EL.16	0/1	—	1/1	II
EEPRO100	EEPRO100_TX.S	4/0	3/0	11/4	—
EEPRO100	MAC_ADDR1.8.A	—	—	6/6	—
EEPRO100	EEPRO100_RX.S	2/0	1/0	6/4	—
PCNET	PCNET_XDA.S	2/0	—	3/2	—
PCNET	PCNET_TMD.S	2/3	1/1	5/5	—
PCNET	PCNET_RDA.S	2/0	—	3/2	—
PCNET	PCNET_RMD.S	3/4	1/0	5/5	—
PCNET	PCNET_INITBLK32.S	10/12	—	13/13	—
PCNET	PCNET_INITBLK16.S	3/10	—	10/10	—
RTL8139	RTL8139_RX_RING_DESC_RXDW0.32	1/1	—	1/1	—
RTL8139	RTL8139_RX_RING_DESC_RXDW1.32	1/0	—	1/1	—
RTL8139	RTL8139_RX_RING_DESC_RXBUFLO.32	—	1/1	1/1	—
RTL8139	RTL8139_RX_RING_DESC_RXBUFHI.32	—	—	1/1	—
RTL8139	RTL8139_TXBUFFER.8.A	—	—	1/1	—
RTL8139	RTL8139_TX_RING_DESC_TXDW0.32	1/1	—	1/1	—
RTL8139	RTL8139_TX_RING_DESC_TXDW1.32	1/1	—	1/1	—
RTL8139	RTL8139_TX_RING_DESC_TXBUFLO.32	—	1/1	1/1	—
RTL8139	RTL8139_TX_RING_DESC_TXBUFHI.32	—	—	1/1	—
EHCI	entry.32	1/1	1/0	1/1	—
EHCI	EHCIqtd.S	3/4	7/7	8/8	—
EHCI	EHCIqh.S	6/7	9/4	12/12	—
EHCI	EHCIitd.S	11/11	8/15	16/16	—
EHCI	EHCIsitd.S	3/1	1/0	7/7	—
OHCI	OHCI_HCCA.S	—	32/32	35/35	—
OHCI	OHCI_ED.S	3/3	3/2	4/4	I
OHCI	OHCI_TD.S	4/4	3/1	4/4	I
OHCI	OHCI_ISO_TD.S	3/3	1/1	12/12	I
UHCI	link.32	1/1	1/1	1/1	—
UHCI	UHCI_QH.S	2/2	2/2	2/2	—
UHCI	UHCI_TD.S	3/3	2/2	4/4	—
XHCI	XHCITRB0.S	2/2	1/3	5/5	—

表 4.4 对于在整个虚拟外设模糊测试的过程中的三个人工步骤，即添加一个新的 VMM、解决静态分析的不确定性以及添加一个新的成组突变器，本表详细说明了必要的手工劳动和估计的平均时间（一周有 40 个工作小时）

Step	Manual effort	Estimated average time
Add a new VMM	Register a virtual device by searching its architecture, the launch command line, and the signature of PIO/MMIO regions.	10 minutes per virtual device
	Initialize a VMM and identify the testing interfaces by following the main() in an existing VMM frontend.	A week per VMM (up to two weeks for debugging)
	Decide and implement the dispatching methods by looking for guest memory access functions.	An hour per VMM
Finish the rest of the annotation extraction after scanning the source code of a virtual device with our static analysis engine	Extract the definition of unnamed types by looking at the source code.	Two minutes per case
	Match two taint analysis results touching the same variable due to disjointed control flow by reading the source code.	15 minutes per case
	Extract the head-tail pointer context by reading the source code.	10 minutes per case
	Extract the flag/tag pointer context by reading the source code.	20 minutes per case
	Extract the length and buffer context by reading the source code.	Five minutes per case
Add a new group mutator	Obtain the insight about what group mutator is necessary by fuzzing virtual devices.	N/A
	Decide the feedback and develop the handler with the help of our action-trigger protocol.	Hours per case (up to two days for debugging)

表 4.3 续上页

XHCI	XHCIEvRingSeg.S	1/0	1/0	3/3	—
XHCI	XHCI_POCTX.64	—	1/1	1/1	—
XHCI	XHCI_CTX.32.A	1/1	—	2/2	—
XHCI	XHCI_SLOT_CTX.32.A	4/4	—	4/4	—
XHCI	XHCI_EP0_CTX.32.A	3/3	—	5/5	—
	Missing (False Negative)	38/128	19/95	27/396	
		29.69%	20.00%	6.82%	
	Wrong (False Positive)	16/128	9/95	4/396	
		12.50%	9.47%	1.01%	

4.6.1 消息内依赖注释提取的结果和人工工作量评估

我们在 18 个 QEMU 虚拟外设上执行我们的推理引擎工具（小节 4.5.1），结果如表 4.3所示。该分析自动提取了 93% 的结构定义，80% 的指针和 70% 的标志位。关于缺失的信息，我们发现有三个原因，与静态分析的局限性有内在联系。

未命名类型。CodeQL2.10.5 不能推断出未命名结构/单元的定义。为了解决这个问题，该工具报告了定义结构/单元的源代码位置，然后依靠人工确认。我们总共发现了 4 个未命名的结构体和 3 个未命名的联合。

状态感知执行。一个虚拟外设的两个不相干的控制流段只有在虚拟外设达到特定的内部状态时才能被连接起来，例如，通过两个或多个消息。在没有具体知识的情况下，CodeQL 的污点分析显示了它在不相连的控制流之间污点传播的局限性。在我们的原型中，我们手动解决这个问题。另一个可行的方案是采用类型敏感的算法[101]，推断出由相同类型的变量控制的控制流段。我们总共观察到六个结构体，涵盖了四个虚拟外设。

上下文感知依赖。静态分析在提取复杂的数据关系（如链表）时很困难，不得不依靠人工分析。在我们的评估中，只有五个虚拟外设（在 18 个中）属于这个类别。

我们的推理引擎对 18 个 QEMU 虚拟外设进行了建模，而 Nyx-SPEC 在 16 个 QEMU 虚拟外设中只为 XHCI 建模。表 4.3详细介绍了结果。此外，为了量化人工努力，表 4.4估计了每种情况下支持注释、新的虚拟机管理程序和成组突变器的平均时间。

表 4.5 6 个模糊测试器的最终代码覆盖率结果，即：VDF、HyperCube、Nyx（Nyx-Spec 只支持 XHCI）、V-Shuttle、QEMUFuzzer 和 ViDeZZo：'—' 表示不支持该虚拟外设；有色的数字报告了我们评估中 10 次运行 24 小时的平均覆盖率，其他数字来自相应的论文

Device	VDF	HyperCube	Nyx-Legacy	V-SHUTTLE	QEMUFuzzer	ViDeZZo
AC97	53.0%	100%	94.04%	—	**95.93%**	95.90%
CS4231a	56.0%	74.76%	75.36%	85.80%	**94.06%**	92.61%
ES1370	72.7%	91.38%	89.69%	91.91%	88.40%	**91.36%**
Intel-HDA	58.6%	79.17%	62.61%	78.30%	**65.87%**	64.78%
SB16	81.0%	83.80%	83.12%	81.52%	84.15%	**87.54%**
AHCI	—	—	—	61.60%	49.89%	**62.06%**
FDC	70.5%	84.51%	**70.06%**	—	69.23%	69.72%
Megasas	—	—	—	58.50%	58.67%	**76.74%**
SDHCI	90.5%	81.15%	**73.58%**	—	71.34%	68.52%
VirtIO-BLK	—	—	—	—	30.55%	**55.39%**
E1000	81.6%	66.08%	53.36%	74.50%	35.32%	**82.27%**
E1000E (1/2)	—	—	—	—	**63.12%**	60.94%
E1000E (2/2)	—	—	—	—	35.48%	**40.84%**
EEPro100	75.4%	83.32%	82.12%	—	82.13%	**90.46%**
NE2000	71.7%	71.89%	74.35%	71.90%	75.09%	**94.00%**
PCNET	36.1%	78.81%	78.87%	88.90%	**93.27%**	92.10%

表 4.5 续上页

RTL8139	63.0%	74.68%	**83.33%**	80.82%	83.06%	77.46%
ATI-VGA (1/2)	—	—	—	79.40%	—	80.69%
ATI-VGA (2/2)	—	—	—	—	—	85.67%
CIRRUS-VGA	—	—	—	—	88.65%	**89.68%**
EHCI	—	—	—	31.19%	71.84%	**71.96%**
OHCI	—	—	—	36.62%	77.33%	**83.99%**
UHCI	—	—	—	22.27%	55.90%	**72.00%**
XHCI	—	64.40%	63.24%	—	52.92%	**81.63%**
			Nyx-Spec			
XHCI			77.12%			
VirtIO-BLK	—	—	—	—	—	55.39%
PL041 (Audio)	—	—	—	—	—	83.91%
SMC91C111 (Net)	—	—	—	—	92.14%	92.98%
TC6393XB (Display)	—	—	—	—	—	76.38%
XLNX-ZYNQMP-CAN	—	—	—	—	—	70.42%
XLNX-DP (Display)	—	—	—	—	—	90.42%
VirtualBox x86_64						
SB16	—	—	—	—	—	61.33%
FDC	—	—	—	—	—	39.32%
PCNET	—	—	—	—	—	48.35%
OHCI	—	—	—	—	—	36.13%

4.6.2 基于依赖感知的模糊测试框架的效率评估

我们说明，ViDeZZo 支持不同类别的虚拟外设，并且与最先进的虚拟外设模糊测试器相比，在可扩展性、最终代码覆盖率、覆盖率随时间变化等多个方面有较高竞争力。

可扩展性。表 4.5显示，ViDeZZo 可以扩展到涵盖五个设备类别、四个架构和两个虚拟机管理程序的 28 个虚拟外设。可扩展性是由三个原因造成的。首先，我们的灵活设计将模糊测试逻辑（ViDeZZo-Core）从 VMM 实现（ViDeZZo-VMM）中抽象出来，从而简化了在不同的虚拟外设、设备类别、架构和管理程序上移植 ViDeZZo-Core 的工作。第二，我们的轻量级注释允许将新的虚拟外设调整到 ViDeZZo。第三，已经注解过的虚拟外设可以在不同的管理程序上进行测试，从而进一步减少移植的工作量。

最终代码覆盖率。对于有两个或更多彩色数字的表4.5中的行，即我们复现相关工作的地方，我们观察到ViDeZZo达到有竞争力的（8/22）甚至更高的（14/22）覆盖率。

- Nyx。Nyx-Legacy对存储设备效果更好，而ViDeZZo对音频、网络和USB设备效果更好。在24小时内，ViDeZZo比Nyx-Spec工作得更好。
- V-Shuttle。对于USB设备，ViDeZZo比V-Shuttle工作得更好。根据实验结果，V-Shuttle（不含种子）不能达到其他工具那样高的覆盖率，原因有两个。首先，覆盖率受到初始语料库的影响。V-Shuttle的作者还提到，初始种子可以进一步提高模糊测试的效率。V-Shuttle使用了BIOS和客户内核初始化虚拟外设时收集的种子。我们推测这些种子编码了intra-message dependencies，提高了最终的覆盖率和速度。相反，我们在复现V-Shuttle结果时没有使用初始种子。ViDeZZo和V-Shuttle之间的比较表明，ViDeZZo产生的种子质量更高，能自主地覆盖更多有趣的测试案例。其次，V-Shuttle的作者确认，他们手动执行了存储和加载VMM的动作，从而添加和删除了一个设备，以覆盖那些否则不会达到的代码，这意味着他们论文中报告的最终结果可能高于没有手动干预的结果。为了使比较公平，我们在复制V-Shuttle结果时没有进行这些操作。
- QEMUFuzzer。对于有更多注释的存储、网络和USB虚拟外设，ViDeZZo比QEMUFuzzer效果更好。我们将QEMUFuzzer的高最终覆盖率归功于其在模糊化过程中对DMA访问模式的支持，因为这些简单的模式可以慢慢猜出部分消息内的依赖关系。
- 重要的是，没有一个模糊测试器可以达到100％的覆盖率，因为有些代码在评估的环境中是永远无法达到的，原因是：(1) 条件性编译标志；(2) 配置错误处理程序；(3) 迁移回调；(4) 清理功能。例如，与QEMU相比，VirtualBox虚拟外设的基本块覆盖率较低，其中一个原因是有更多无法到达的函数。

覆盖率随时间的变化。图4.13显示，在最初的十秒钟内，ViDeZZo的覆盖率急剧增加，然后在一小时内达到一个平稳的状态。尽管Nyx-Legacy在开始时有更高的覆盖率，但ViDeZZo在几秒钟内就追上了。总的来说，ViDeZZo比Nyx-Legacy、QEMUFuzzer和V-Shuttle更快地达到了有竞争力的最终覆盖率结果。有趣的是，与较浅的代码（图4.13l）相比，具有较深的调用栈（图4.13k）的代码探索得更慢。

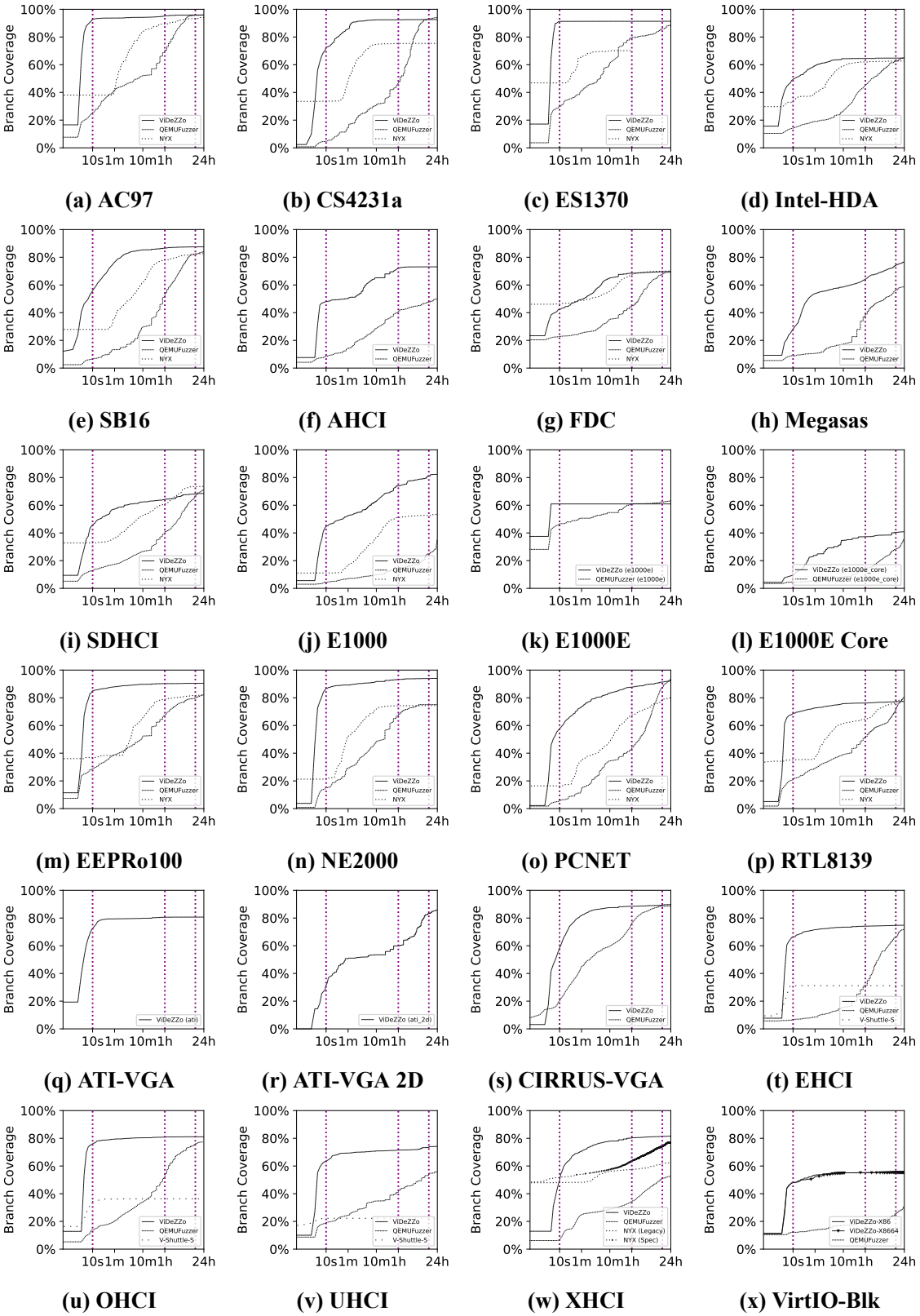


图 4.13 由 Nyx、V-Shuttle、QEMUFuzzer 和 ViDeZZo 模糊测试的虚拟外设 24 小时内的完整版本覆盖率：阴影显示最小和最大的覆盖率，黑线显示平均覆盖率

表 4.6 ViDeZZo 的变体和基线汇总：具体来说，ViDeZZo-ARP 保留了我们所有的设计选项和技术，也就是说，intrA-message annotation（A），inteR-message mutators（R）和Persistent fuzzing（P）；此外，我们还测试了 QEMUFuzzer、V-SHUTTLE、Nyx-Spec 和 ViDeZZo++

Variants	IntrA-Message Dependency	InteR-Message Dependency	Persistent Fuzzing
ViDeZZo-ARP	✓	✓	✓
ViDeZZo-AP	✓	N/A	✓
ViDeZZo-RP	N/A	✓	✓
ViDeZZo-P	N/A	N/A	✓
QEMUFuzzer	N/A	N/A	N/A
V-SHUTTLE	N/A	N/A	N/A
Nyx-Spec	✓	✓	N/A
ViDeZZo++-ARP	✓	✓	✓

```
while (1) {
  TRBType type;
  pci_dma_read(pci_dev, ring->dequeue, trb, TRB_SIZE);
+ __sanitizer_cov_trace_state(0, 1);
  trb->addr = ring->dequeue;
  trb->ccs = ring->ccs;
```

图 4.14 QEMU XHCI 的状态反馈插装示意图

4.6.3 系统设计中关键部分的作用评估

我们在覆盖率和开销方面对我们的设计选择进行了敏感性分析。具体来说，我们设计了四个 ViDeZZo 变体，其后缀表示启用的功能，即：intrA-message annotation（A），inteR-message mutators（R），和Persistent fuzzing（P）。例如，ViDeZZo-ARP 显示了启用所有三个功能后的结果。ViDeZZo-AP 没有消息间突变器，但在 libFuzzer 中启用了字节突变功能。一般来说，这些变体显示了每个特性的独立和组合的影响。

表 4.6显示了该设置。我们选择 QEMUFuzzer 和 V-Shuttle 作为基线，并包括 Nyx-Spec 和 ViDeZZo++ 来评估状态感知。我们基于 ViDeZZo-ARP 实现了 ViDeZZo++，它包含一个类似于 FuzzUSB 的简单状态感知机制。ViDeZZo 和 QEMUFuzzer 支持所有四个控制器，V-Shuttle 不支持 XHCI，而 Nyx-Spec 和 ViDeZZo++ 只支持 XHCI。

ViDeZZO++ 的实现。 ViDeZZo++ 模仿 FuzzUSB 来支持 QEMU XHCI 的状态和状态转换反馈（目前 ViDeZZo++ 只支持 QEMU XHCI）。首先，像 FuzzUSB 一样，状态转换点被定义为一个 DMA 访问，例如，pci_dma_read()。然后，每当一个状态转换点被访问，虚拟外设就会进入一个新的状态。接下来，我们用 __sanitizer_cov_trace_state() 手动检测 QEMU XHCI，其第一个参数对 QEMU XHCI 来说总是 0，第二个参数是新状态

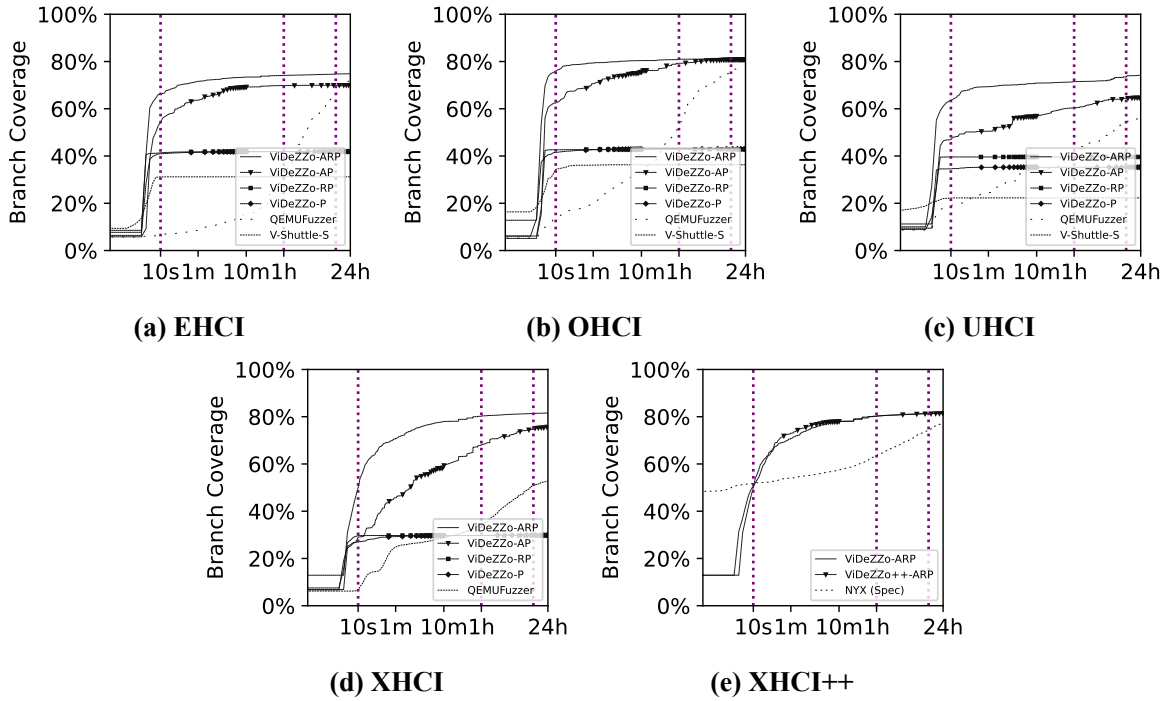
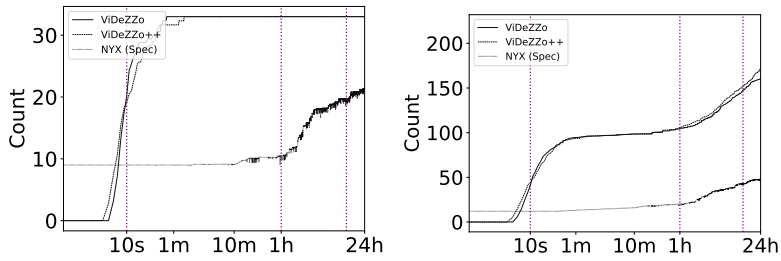


图 4.15 ViDeZZo 变体和基线的分支代码覆盖率结果

的标识符，以明确地将过渡点后的新状态传递给 libFuzzer（图 4.14）。我们实现了两个字节图（第一个是一维的状态，另一个是二维的状态转换）来计算一个状态或一个状态转换被遇到多少次（最多 255 次）。此外，这两个字节图被纳入 libFuzzer 的特征集合。如果涵盖任何新的状态或状态转换，这个种子将被添加到语料库中。

有效的依赖感知的模糊测试。图 4.15提出了四个方面的发现。第一，ViDeZZo-ARP 和 ViDeZZo-AP 之间的差异表明我们的消息间突变器对新的覆盖率和覆盖速度都有贡献。第二，ViDeZZo-ARP 和 ViDeZZo-RP 之间的差异表明我们的消息内注释对新覆盖率的影响更大。第三，ViDeZZo-ARP、ViDeZZo-RP 和 ViDeZZo-P 之间的差异表明，当启用消息内注释时，我们的消息间突变器更加有效。第三个观察结果是意料之中的，因为在没有消息内依赖的情况下，模糊测试器是在浅层代码空间中打转的。第四，ViDeZZo 和 ViDeZZo++ 之间的微小差异表明，状态感知并没有显著提高代码覆盖率。然而，在图 4.16b中显示，ViDeZZo++ 发现了不同的状态转换（路径），即不同的覆盖率，提出了一个发现新安全缺陷的探索角度。

开销。表 4.7显示了每秒钟的执行情况。在这里，我们观察到三个发现。首先，ViDeZZo 通常比 QEMFUzzer 快。第二，消息内注释的性能似乎取决于目标。第三，由于每次迭代的突变次数减少，消息间突变器的速度提高了 4 到 5 倍。



(a) 状态覆盖率　　**(b) 状态转移覆盖率**

图 4.16 24 小时内的状态和状态转移的覆盖率结果

表 4.7 24 小时内 ViDeZZo 变种的开销结果

Fuzzer	EHCI	OHCI	UHCI	XHCI
	Average Speed (exec/s)			
ViDeZZo-ARP	3679.18	763.77	4263.97	4350.51
ViDeZZo-AP	1135.82	198.69	850.31	1111.21
ViDeZZo-RP	3221.41	6735.11	3006.41	5393.13
ViDeZZo-P	1135.20	1358.38	820.81	1136.90
QEMUFuzzer	120.38	141.70	93.27	169.54

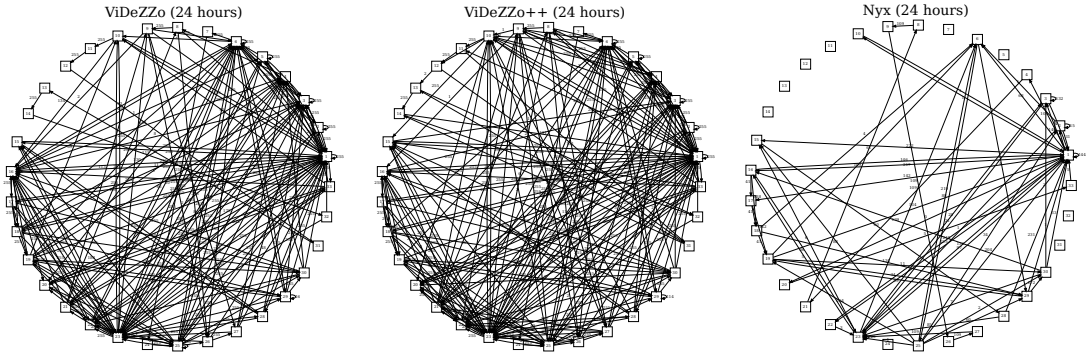
状况覆盖率。在下文中，我们展示了 Nyx-Spec、ViDeZZo 和 ViDeZZo++ 的状态（例如，QEMU XHCI 有 33 个不同状态）和状态转换的数量（状态转换空间为 33×33）。如图 4.16a所示，ViDeZZo 和 ViDeZZo++ 覆盖状态的速度比 Nyx-Spec 快。相反，在图 4.16b中，ViDeZZo++ 的表现略好于 ViDeZZo，这是预期的，因为前者是状态感知的。最终的状态机如图 4.17a、图 4.17b和图 4.17c所示。

4.6.4　复现已有漏洞的能力评估

表 4.8比较了 V-Shuttle、QEMUFuzzer 和 ViDeZZo 如何发现在 V-Shuttle 论文中列出的五个漏洞。我们在最多 24 小时内执行每个试验 10 次，然后对执行次数进行平均，以发现漏洞。最初，我们未能在 24 小时内重现 CVE-2020-25085，因为没有注释一个 12 位子字段。我们通过手动注释子字段来解决这个问题，从而允许 ViDeZZo 绕过该块。

4.6.5　长时间运行下的漏洞挖掘能力评估

我们运行 ViDeZZo 24 小时，在 QEMU 6.1.50 及以上版本和 VirtualBox 7.0.0 上成功重现 24 个现有安全缺陷，并发现 28 个新的安全缺陷（表 4.9）。具体来说，我们对每个虚拟外设进行 24 小时的模糊测试，每个虚拟外设使用一个 CPU。只要有崩溃出现，我们就



(a) ViDeZZo **(b) ViDeZZo++** **(c) Nyx-Spec**

图 4.17 24 小时后的状态和状态转移分布示意图

表 4.8 虚拟外设模糊测试器触发漏洞所需的平均执行次数，以及最小和最大执行次数（"–" 表示缺乏支持；有色的数字来自我们的评估，无色的数字来自相应的论文；IO 是整数溢出，UAF 是释放后使用，HBO 是堆缓冲区溢出，IL 是无限循环）

Bug	Description	V-Shuttle	QEMFUzzer	ViDeZZo
CVE-2020-11869	ATI-VGA IO	35.6M	—	782K (98.0K–2.85M)
CVE-2020-25084	EHCI UAF	79.4M	1.80M (1.36–2.23M)	44.0M (11.7M–88.8M)
CVE-2020-25085	SDHCI HBO	8.88M	1.58M (1.28M–1.85M)	32.3M (1.74M–114M)
CVE-2020-25625	OHCI IL	40.5M	TIMEOUT	2.22K (1.02K–6.22K)
CVE-2021-20257	E1000 IL	235K	TIMEOUT	283K (101K–618K)

把它视为一个安全缺陷，进行判断，并报告。此外，我们还积极与 QEMU 和 VirtualBox 社区合作，对现有的和新发现的安全缺陷进行修补。

漏洞影响力分析。表 4.9表明，虚拟外设有不同类型的安全缺陷，不仅包括断言失败和中止，还包括空间和时间上的内存损坏。断言失败和中止有 59.61% (31/52)，因 Ubuntu 在其 QEMU 二进制中启用了断言功能，这些断言失败和中止能在 Ubuntu 上造成拒绝服务。我们获得了一个 CVE，并提供了 7 个补丁，都已被合并到代码仓中。虚拟外设中的安全缺陷很常见，而且其多样性丰富了攻击原语并降低了攻击难度。

下面的两个案例研究显示了为什么消息内注释和消息间突变器很重要。

案例研究 1：触发了 QEMU EHCI 控制器中的一个先前已知的用户释放后问题（CVE-2020-25084）。有了一个最小化的 PoC 之后，我们得出了以下两个结论。消息间的突变器自动找到了一个 "写和等待" 的消息序列模式。PoC 的第一阶段用多个写消息设置了 EHCI 的内部状态。该序列以一个 cloc_step 消息结束，该消息推进了系统时间，迫使所有定时器到期，并激活了异步的 ehci_work_bh() 来处理 USB 数据包。这种 "写和等待"

的模式在驱动开发中很常见。利用消息内依赖注释影响了控制流，导致了 UAF 错误。PoC 的第二阶段从物理内存加载一个缓冲区。EHCI 根据之前的内部状态和缓冲区的值达到不同的状态。然后，EHCI 试图通过 usb_packet_map() 将一个 USB 数据包映射到主机地址空间，但是失败了。错误处理程序，即 usb_packet_unmap()，回滚之前的映射。然而，EHCI 并没有同步映射失败的情况。因此，usb_packet_unmap() 被再次调用，从而导致了一个释放后使用的错误。

案例研究 2：通过对 QEMU OHCI 控制器进行模糊测试，触发了 QEMU USB 存储设备中一个之前未知的不一致的安全缺陷。在该存储设备中，USB_MSDM_DATATOUT 与 SCSI_XFER_FROM_DEV 相冲突，这种不一致引发了存储设备的断言失败。我们构建了一个最小化的 PoC，受益于消息内注释和消息间突变器：(1) 消息内注释有助于通过关键的约束检查。具体来说，ViDeZZo 通过标志和指针字段对 OHCI[102]的端点描述符链和传输描述符链进行建模，保证 OHCI 遍历描述符并允许存储设备访问描述符本身内部的数据。使用约束条件使 ViDeZZo 产生一个正确的 SCSI 命令，状态为 SCSI_XFER_FROM_DEV。(2) 消息间突变器选择一个特定的消息序列，允许存储设备达到错误状态。特别是，存储设备必须跨越从 USB_MSDM_CBW 到 USB_MSDM_DATAOUT 的多个阶段，这是通过消息间突变器来实现的，这些突变器逐渐学习了消息顺序。

表 4.9 QEMU 和 VirtualBox 的安全缺陷汇总（“# of Messages” 一栏是在差分调试的帮助下触发这个安全缺陷所需最少的消息数量，如果该安全缺陷已被修复，我们就不做差分调试并标记 “N/A”；对于 “Reported-By ” 列，如果是以前已知的错误，我们列出触发该错误的工具名称，即：Nyx (Ny), VShuttle (VS), QEMUFuzzer (QF), 和 ViDeZZo (Vi) ，我们使用 Anonymous (An) 代表未知的工具；对于 “状态” 一栏，如果一个安全缺陷没有补丁，我们标记为 “Open”，如果我们提交了一个补丁，我们标记为 “Patch submitted”，如果一个安全缺陷已经被修复，我们没有参与，我们标记为 “Fixed”，如果我们参与，我们标记为 “$\widetilde{Fixed}$”，特别是，如果我们的补丁被接受，我们标记为 “$\widetilde{Fixed(us)}$”）

Id	Target	Category	VMM	Version	Arch	Short Description	# of Messages	Reported By	Status
1	ac97	audio	qemu	7.0.94	i386	Abort in audio_calloc()	1	An, Vi	Fixed
2	am53c974	storage	qemu	6.1.50	i386	Null pointer access in do_busid_cmd()	N/A	An	Fixed
3	ati	display	qemu	6.1.50	i386	Out of bounds write in ati_2d_blt()	N/A	VS	Fixed
4	ati	display	qemu	7.0.94	i386	Out of bounds write in ati_2d_blt()	4	Vi	$\widetilde{Fixed}$
5	cadence_uart	serial	qemu	7.2.50	aarch64	Devision by zero in uart_parameters_setup()	2	Vi	$\widetilde{Fixed}$
6	dwc2	usb	qemu	6.1.50	aarch32	Assertion failed in hw/usb/core.c from dwc2	N/A	Vi	Patch submitted
7	e1000	net	qemu	6.1.50	i386	Infinite loop in process_tx_desc()	N/A	Ny, VS, QF	Fixed
8	ehci	usb	qemu	6.1.50	i386	Abort in usb_ep_get()	N/A	Ny, VS, QF	Patch submitted
9	ehci	usb	qemu	6.1.50	i386	Assertion failure in address_space_unmap()	N/A	Ny, VS, QF	Fixed
10	exynos421_fimd	display	qemu	7.2.50	aarch32	Assertion failure in fimd_update_memory_section()	2	Vi	Open
11	ftgmac100	net	qemu	7.2.50	aarch32	Heap buffer overflow in aspeed_smc_flash_do_select()	2	Vi	Open
12	imx_usb_phy	usb	qemu	7.2.50	aarch32	Out of bounds in imx_usbphy_read()	1	Vi	$\widetilde{Fixed}$
13	intel-hda	audio	vbox	7.0.7	i386	Global buffer overflow in hdaMmioWrite()	1	Vi	Open
14	lan9118	net	qemu	7.2.50	aarch32	Out of bounds read in lan9118	535	Vi	Open
15	lan9118	net	qemu	7.2.50	aarch32	Abort in lan9118_16bit_mode_read()	1	Vi	$\widetilde{Fixed(us)}$
16	lsi53c895a	storage	qemu	6.1.50	i386	Assertion failure in lsi53c810 emulator	N/A	Ny, VS, QF, An, Vi	Fixed
17	megasas	storage	qemu	6.1.50	i386	Assertion failure in scsi_dma_complete()	N/A	QF	Fixed
18	megasas	storage	qemu	6.1.50	i386	Assertion failure in bdrv_co_write_req_prepare()	N/A	QF	Fixed
19	nvme	storage	qemu	6.1.50	i386	Null pointer access in memory_region_set_enabled()	1	Vi	$\widetilde{Fixed(us)}$
20	ohci	usb	qemu	7.0.50	i386	Assertion failure in usb_msd_transfer_data()	29	Vi	$\widetilde{Fixed}$
21	ohci	usb	qemu	7.0.50	i386	Abort in ohci_frame_boundary()	8	VS, QF, Vi	$\widetilde{Fixed(us)}$

表 4.9 续上页

22	ohci	usb	qemu	7.0.50	i386	Heap use after free in usb_cancel_packet()	67	Vi	Open
23	ohci	usb	qemu	7.0.91	i386	Assertion failure in usb_cancel_packet()	79	An	Open
24	omap_dss	display	qemu	7.2.50	aarch32	Out of memory in hw/omap-dss for aarch32	3	Vi	Open
25	pl041	audio	qemu	7.0.94	aarch32	Abort in audio_bug() triggered by pl041	1	An	Fixed
26	pl041	audio	qemu	7.0.94	aarch32	Abort in audio_bug() triggered by pl041	2	An	Fixed
27	sb16	audio	qemu	6.1.50	i386	Assertion failure in audio_calloc() caused by sb16	N/A	An	Fixed
28	sb16	audio	qemu	6.1.50	i386	Abort in audio_calloc()	4	Vi	$\widetilde{Fixed(us)}$
29	sdhci	storage	qemu	7.1.50	i386	Heap buffer overflow in sdhci_read_dataport()	9	QF	Fixed
30	smc91c111	net	qemu	7.1.93	aarch32	Out of bounds read/write in smc91c111	5	Vi	Open
31	tc6393xb	display	qemu	7.2.50	aarch32	negative-size-param in nand_blk_load_512()	23	Vi	Open
32	tc6393xb	display	qemu	7.2.50	aarch32	Heap buffer overflow in nand_blk_write_512()	7	Vi	Open
33	virtio-blk	storage	qemu	7.0.94	i386	Assertion failure in address_space_stw_le_cached()	5	An	Fixed
34	virtio-blk	storage	qemu	7.0.94	i386	Infinite loop in virtio_blk_handle_vq()	16	An	Fixed
35	vmxnet3	net	qemu	6.1.50	i386	Code should not be reached vmxnet3_io_bar1_write()	N/A	VS, Vi	Fixed
36	vmxnet3	net	qemu	6.1.50	i386	Three hw_error() in vmxnet3_validate_queues()	N/A	QF	Fixed
37	vmxnet3	net	qemu	6.1.50	i386	Assertion failed in vmxnet3_io_bar0_write()	N/A	QF	Fixed
38	vmxnet3	net	qemu	6.1.50	i386	Out of memory net_tx_pkt_init()	N/A	QF, VS	Fixed
39	vmxnet3	net	qemu	6.1.50	i386	Assertion failure in net_tx_pkt_reset()	N/A	QF	Fixed
40	vmxnet3	net	qemu	6.1.50	i386	eth_get_gso_type: code should not be reached	N/A	QF, VS	Fixed
41	xhci	usb	qemu	7.0.94	i386	Abort in xhci_find_stream()	56	Vi	$\widetilde{Fixed(us)}$
42	xlnx_dp	display	qemu	7.0.91	aarch64	Abort in xlnx_dp_aux_set_command()	1	Vi	$\widetilde{Fixed(us)}$
43	xlnx_dp	display	qemu	6.1.50	aarch64	Out of bounds read in xlnx_dp_read()	1	Vi	$\widetilde{Fixed(us)}$
44	xlnx_dp	display	qemu	6.1.50	aarch64	Out of bounds in xlnx_dp_vblend_read()	N/A	An	Fixed
45	xlnx_dp	display	qemu	7.2.50	aarch64	Overflow in xlnx_dp_aux_push_rx_fifo()	3	Vi	Patch submitted
46	xlnx_dp	display	qemu	7.2.50	aarch64	Abort in xlnx_dp_change_graphic_fmt()	1	Vi	Patch submitted
47	xlnx_dp	display	qemu	7.2.50	aarch64	Underflow in xlnx_dp_aux_pop_tx_fifo()	1	Vi	Patch submitted
48	xlnx_dp	display	qemu	7.2.50	aarch64	Overflow in xlnx_dp_aux_push_tx_fifo()	17	Vi	Patch submitted
49	zynqmp_can	net	qemu	7.2.50	aarch64	Fifo underflow in transfer_fifo()	2	Vi	Open
50	zynqmp_can	net	qemu	7.2.50	aarch64	Fifo overflow in transfer_fifo()	291	Vi	Open
51	zynqmp_qspips	spi	qemu	7.2.50	aarch64	Out of bound in xilinx_spips_write()	1	Vi	Open
52	zynqmp_qspips	spi	qemu	7.2.50	aarch64	Underflow in xlnx_dp_aux_push_rx_fifo()	2	Vi	Open

4.7 本章小结

虚拟外设的模糊测试必须跟踪消息内和消息间的依赖关系。我们为虚拟外设提出了一个依赖感知的模糊测试框架 ViDeZZo，结合了这两种依赖关系。在这个框架中，我们提出了一个轻量级的语法和三类新的消息突变器，共同促进了模糊测试的效率的提升。与之前的工作相比，ViDeZZo 既具有可扩展性（涵盖了两个虚拟机管理程序、四个架构、五个设备类别和虚拟外设），又具有高效性（更快地达到了可比的代码覆盖率）。我们成功地复现了 24 个现有的错误，并发现了 28 个新的错误，获得了一个 CVE，涵盖了多样的安全缺陷类型。我们提供了 7 个补丁，已经合并到代码仓中。

5 总结和展望

5.1 本文工作总结

为了实现 Linux 物联网设备虚拟化的两个核心目标，本文提出了基于模型引导内核执行的 Linux 物联网设备虚拟执行环境构建方法，实现了虚拟执行环境的高保真性；和基于依赖感知消息模型的虚拟机管理程序模糊测试方法，实现了虚拟执行环境的高安全性。本文所提技术结合起来，突破了已有技术的瓶颈，促进了 Linux 物联网设备安全研究的开展和应用，如漏洞挖掘（模糊测试）和漏洞分析等。本文有以下结论。

第一，构建 Linux 物联网设备的虚拟执行环境，一种可行的方案是考虑“最小努力交付”，将外围设备分成 I-型和 II-型外围设备，并采取不同的策略。原型系统 FirmGuide 生成了 9 个具有完整功能的 I-型虚拟外设和 64 个具有最小功能的 II-型虚拟外设，支持了 26 个片上系统。用从互联网上下载的 Linux 物联网设备的固件进行的实验表明，FirmGuide 成功地重新托管超过 95% 的 Linux 内核，涵盖了两个架构和 22 个内核版本；最终，成功地支持了 Linux 物联网设备的漏洞挖掘（模糊测试）和漏洞分析。

第二，模糊测试虚拟机管理程序中的虚拟外设需要考虑虚拟外设消息内依赖和消息间依赖：解决消息内依赖，可以采取轻量级的消息注释；解决消息间依赖，可以采用精心设计的突变器。原型系统 ViDeZZo 高效地和大规模地测试了虚拟机管理程序中的虚拟外设。与之前的工作相比，ViDeZZo 既具有可扩展性（涵盖了两个虚拟机管理程序、四个架构、五个设备类别和 28 个虚拟外设），又具有高效性（更快地达到了可比的代码覆盖率）；成功地复现了 24 个已报道的安全缺陷，发现了 28 个新的安全缺陷，获得了一个 CVE 。此外，7 个补丁已经合并到代码仓主线中。

5.2 未来工作展望

虽然本文已经实现了高保真度和高安全性的 Linux 物联网设备虚拟执行环境，但在虚拟执行环境构建和虚拟执行环境测试上仍有提高空间，具体如下。

第一，如何对 II-型外围设备进行功能级别的建模？模型引导内核执行对 Linux 物联网设备模糊测试、漏洞分析已经足够，但是如果想将重新托管的 Linux 物联网设备以蜜

罐的方式部署在网络上，需要进一步处理 II-型外围设备，以免被进入蜜罐的攻击者发现自己在一个虚拟执行环境中。ECMO[19] 通过动态二进制改写将与网卡驱动的交互导出到一个已虚拟化的网卡驱动上，类似于“体外人工肺”，是一种可行的解决方案。但是替换网卡驱动是一种间接的方法，并没有直接解决这个问题。假设能对 II-型外围设备直接建模，其模型甚至也能应用在虚拟设备的模糊测试上。

第二，如何通过反馈机制引导模糊测试探索深层次的程序空间？在对虚拟机管理程序的虚拟外设进行模糊测试时，覆盖率常常集中在距离客户机接近的区域，与宿主机接近的区域覆盖较少。在输入生成器尚可的前提下，以上现象说明，目前的反馈机制已经不能满足探索深层次的程序空间的需求。基于状态感知的模糊测试是一种可行的解决方案：将整个程序从逻辑上划分不同的状态，将状态空间的覆盖率作为优先级较高的反馈机制指导模糊测试，从全局优化的角度避免陷入局部最优。

解决以上两个问题将进一步提高 Linux 物联网设备虚拟执行环境的保真度和安全性，有助于促进相关技术的落地和产业应用。

参考文献

[1] Contributors W. Internet of things[EB/OL]. Wikimedia Foundation. 2023 [2023-08-22]. https://en.wikipedia.org/wiki/Internet_of_things.

[2] Dang F, Li Z, Liu Y, et al. Understanding fileless attacks on linux-based iot devices with honeycloud [C]//Annual International Conference on Mobile Systems, Applications, and Services (MobiSys). 2019: 482-493.

[3] Li H, Huang Q, Ding F, et al. Understanding and Detecting Remote Infection on Linux-based IoT Devices[C]//ACM on Asia Conference on Computer and Communications Security (AsiaCCS). 2022: 873-887.

[4] Angelakopoulos I, Stringhini G, Egele M. FirmSolo: Enabling dynamic analysis of binary Linux-based IoT kernel modules[C]//USENIX Security Symposium (USENIX Security). 2023.

[5] Pustogarov I, Wu Q, Lie D. Ex-vivo dynamic analysis framework for Android device drivers[C]//IEEE Symposium on Security and Privacy (S&P). 2020: 1088-1105.

[6] Liu Q, Zhang C. cyruscyliu/firmguide: Source code of FirmGuide.[EB/OL]. 2021. https://github.com/cyruscyliu/firmguide.

[7] ViDeZZo: Dependency-aware Virtual Device Fuzzing Framework[EB/OL]. GitHub. 2023 [2023-07-02]. https://github.com/HexHive/ViDeZZo.

[8] Antonakakis M, April T, Bailey M, et al. Understanding the Mirai Botnet[C]//USENIX Security Symposium (USENIX Security). 2017: 1093-1110.

[9] Zaddach J, Bruno L, Francillon A, et al. AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares[C]//Network and Distributed System Security Symposium (NDSS). 2014.

[10] Kammerstetter M, Platzer C, Kastner W. Prospect: peripheral proxying supported embedded code testing[C]//ACM Symposium on Information, Computer and Communications Security (AsiaCCS). 2014: 329-340.

[11] Koscher K, Kohno T, Molnar D. SURROGATES: Enabling near-real-time dynamic analyses of embedded systems[C]//USENIX Workshop on Offensive Technologies (WOOT). 2015: 135-150.

[12] Gustafson E, Muench M, Spensky C, et al. Toward the Analysis of Embedded Firmware through Automated Re-hosting[C]//International Symposium on Research in Attacks, Intrusions and Defenses (RAID). 2019: 135-150.

[13] Feng B, Mera A, Lu L. P2IM: Scalable and Hardware-independent Firmware Testing via Automatic

Peripheral Interface Modeling[C]//. 2019: 1237-1254.

[14] Clements A A, Gustafson E, Scharnowski T, et al. HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation[C]//USENIX Security Symposium (USENIX Security). 2020: 1201-1218.

[15] Cao C, Guan L, Ming J, et al. Device-Agnostic Firmware Execution is Possible: A Concolic Execution Approach for Peripheral Emulation[C]//Annual Computer Security Applications Conference (ACSAC). 2020: 746-759.

[16] Spensky C, Machiry A, Redini N, et al. Conware: Automated Modeling of Hardware Peripherals[C]//ACM Asia Conference on Computer and Communications Security (AsiaCCS). 2021: 95-109.

[17] Mera A, Feng B, Lu L, et al. DICE: Automatic emulation of dma input channels for dynamic firmware analysis[C]//IEEE Symposium on Security and Privacy (S&P). 2021: 1938-1954.

[18] Li W L, Guan L, Lin J, et al. From Library Portability to Para-rehosting: Natively Executing Microcontroller Software on Commodity Hardware[C]//Network and Distributed System Security Symposium (NDSS). 2021.

[19] Jiang M, Ma L, Zhou Y, et al. ECMO: Peripheral transplantation to Rehost embedded Linux kernels [C]//ACM Conference on Computer and Communications Security (CCS). 2021: 734-748.

[20] Zhou W, Guan L, Liu P, et al. Automatic Firmware Emulation through Invalidity-guided Knowledge Inference[C]//USENIX Security Symposium (USENIX Security). 2021: 2007-2024.

[21] Johnson E, Bland M, Zhu Y, et al. Jetset: Targeted Firmware Rehosting for Embedded Systems[C]//USENIX Security Symposium (USENIX Security). 2021: 321-338.

[22] Won J Y, Wen H, Lin Z. What You See is Not What You Get: Revealing Hidden Memory Mapping for Peripheral Modeling[C]//International Symposium on Research in Attacks, Intrusions, and Defenses (RAID). 2022: 200-213.

[23] Zhou W, Zhang L, Guan L, et al. What Your Firmware Tells You Is Not How You Should Emulate It: A Specification-Guided Approach for Firmware Emulation[C]//ACM Conference on Computer and Communications Security (CCS). 2022: 3269-3283.

[24] Chen Z, Thomas S L, Garcia F D. MetaEmu: An Architecture Agnostic Rehosting Framework for Automotive Firmware[C]//ACM Conference on Computer and Communications Security (CCS). 2022: 515-529.

[25] Scharnowski T, Bars N, Schloegel M, et al. Fuzzware: Using Precise MMIO Modeling for Effective Firmware Fuzzing[C]//USENIX Security Symposium (USENIX Security). 2022: 1239-1256.

[26] Chesser M, Nepal S, Ranasinghe D C. Icicle: A re-designed emulator for grey-box firmware fuzzing [C]//ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA). 2023.

[27] Farrelly G, Chesser M, Ranasinghe D C. Ember-IO: Effective Firmware Fuzzing with Model-Free Memory Mapped IO[C]//ACM Symposium on Information, Computer and Communications Security (AsiaCCS). 2023: 401-414.

[28] Wu Y, Zhang T, Jung C, et al. DEVFUZZ: Automatic Device Model-Guided Device Driver Fuzzing [C]//IEEE Symposium on Security and Privacy (S&P). 2023.

[29] Clements A A, Carpenter L, Moeglein W, et al. Is your firmware real or re-hosted?[C]//Workshop on Binary Analysis Research (BAR). 2021.

[30] Chen D D, Woo M, Brumley D, et al. Towards Automated Dynamic Analysis for Linux-based Embedded Firmware[C]//Network and Distributed System Security Symposium (NDSS): vol. 16. 2016.

[31] Costin A, Zarras A, Francillon A. Automated dynamic firmware analysis at scale: a case study on embedded web interfaces[C]//ACM on Asia Conference on Computer and Communications Security (AsiaCCS). 2016: 437-448.

[32] Srivastava P, Peng H, Li J, et al. FirmFuzz: Automated IoT Firmware Introspection and Analysis [C]//International ACM Workshop on Security and Privacy for the Internet-of-Things (IoT S&P). 2019: 15-21.

[33] Zheng Y, Davanian A, Yin H, et al. FIRM-AFL: high-throughput greybox fuzzing of iot firmware via augmented process emulation[C]//USENIX Security Symposium (USENIX Security). 2019: 1099-1114.

[34] Therealsaumil. ARM-X Firmware Emulation Framework[EB/OL]. 2019. https://github.com/therealsaumil/armx.

[35] Kim M, Kim D, Kim E, et al. FirmAE: Towards Large-Scale Emulation of IoT Firmware for Dynamic Analysis[C]//Annual Computer Security Applications Conference (ACSAC). 2020: 733-745.

[36] Nico R. Emulation and Exploration of BCM WiFi Frame Parsing using LuaQEMU[EB/OL]. 2017. https://comsecuris.com/blog/posts/luaqemu_bcm_wifi/.

[37] VMSA-2022-0033[EB/OL]. 2022. https://www.vmware.com/security/advisories/VMSA-2022-0033.html.

[38] VMSA-2022-0004[EB/OL]. 2022. https://www.vmware.com/security/advisories/VMSA-2022-0004.html.

[39] VMware Patches VM Escape Flaw Exploited at Geekpwn Event[EB/OL]. 2022. https://www.securityweek.com/vmware-patches-vm-escape-flaw-exploited-geekpwn-event/.

[40] Contributors O F. OSS-Fuzz - continuous fuzzing for open source software[EB/OL]. GitHub. 2022 [2022-07-01]. https://github.com/google/oss-fuzz.

[41] Contributors S. syzbot[EB/OL]. Google. 2023 [2023-07-09]. https://syzkaller.appspot.com/upstrea

m.

[42] Zhu X, Wen S, Camtepe S, et al. Fuzzing: A Survey for Roadmap[J]. ACM Computing Surveys, 2022, 54(11s): 230:1-230:36.

[43] AddressSanitizer[EB/OL]. GitHub. 2019 [2023-07-02]. https://github.com/google/sanitizers/wiki/AddressSanitizer.

[44] The kernel address sanitizer (kasan)[EB/OL]. Linux Kernel. 2014 [2023-07-02]. https://www.kernel.org/doc/html/v4.14/dev-tools/kasan.html.

[45] The AFL++ fuzzing framework | AFLplusplus[EB/OL]. AFLplusplus. 2017 [2023-06-30]. https://aflplus.plus/.

[46] Contributors L. libFuzzer –a library for coverage-guided fuzz testing[EB/OL]. Llvm.org. 2017 [2021-10-06]. https://llvm.org/docs/LibFuzzer.html.

[47] Security oriented software fuzzer[EB/OL]. GitHub. 2023 [2023-07-01]. https://github.com/google/honggfuzz.

[48] Contributors S. Syzkaller is an unsupervised coverage-guided kernel fuzzer[EB/OL]. GitHub. 2021 [2021-10-06]. https://github.com/google/syzkaller.

[49] Bulekov A, Das B, Hajnoczi S, et al. MORPHUZZ: Bending (Input) Space to Fuzz Virtual Devices [C]//USENIX Security Symposium (USENIX Security). 2022: 1221-1238.

[50] Ormandy T. An empirical study into the security exposure to hosts of hostile virtualized environments [C]//CanSecWest. 2007: 1-18.

[51] Yu T, Qu X, Cohen M B. VDTEST: An automated framework to support testing for virtual devices [C]//IEEE/ACM International Conference on Software Engineering (ICSE). 2016: 583-594.

[52] Tang J, Li M. When virtualization encounter AFL[EB/OL]. Black Hat Europe. 2016. https://www.blackhat.com/docs/eu-16/materials/eu-16-Li-When-Virtualization-Encounters-AFL-A-Portable-Virtual-Device-Fuzzing-Framework-With-AFL.pdf.

[53] Schumilo S, Aschermann C, Abbasi A, et al. HYPER-CUBE: High-Dimensional Hypervisor Fuzzing [C]//Network and Distributed System Security Symposium (NDSS). 2020.

[54] Schumilo S, Aschermann C, Abbasi A, et al. NYX: Greybox hypervisor fuzzing using fast snapshots and affine types[C]//USENIX Security Symposium (USENIX Security). 2021: 2597-2614.

[55] Pan G, Lin X, Zhang X, et al. V-SHUTTLE: Scalable and Semantics-Aware Hypervisor Virtual Device Fuzzing[C]//ACM Conference on Computer and Communications Security (CCS). 2021: 2197-2213.

[56] Bulekov A, Das B, Hajnoczi S, et al. MORPHUZZ: Bending (Input) Space to Fuzz Virtual Devices [C/OL]//USENIX Security Symposium (USENIX Security). 2022: 1221-1238. https://www.useni

x.org/conference/usenixsecurity22/presentation/bulekov.

[57] Myung C, Lee G, Lee B. MundoFuzz: Hypervisor Fuzzing with Statistical Coverage Testing and Grammar Inference[C]//USENIX Security Symposium (USENIX Security). 2022: 1257-1274.

[58] Corina J, Machiry A, Salls C, et al. DIFUZE: Interface aware fuzzing for kernel drivers[C]//ACM Conference on Computer and Communications Security (CCS). 2017: 2123-2138.

[59] Pailoor S, Aday A, Jana S. MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation [C]//USENIX Security Symposium (USENIX Security). 2018: 729-743.

[60] Kim K, Jeong D R, Kim C H, et al. HFL: Hybrid Fuzzing on the Linux Kernel.[C]//Network and Distributed System Security Symposium (NDSS). 2020.

[61] Chen W, Wang Y, Zhang Z, et al. Syzgen: Automated generation of syscall specification of closed-source macos drivers[C]//ACM Conference on Computer and Communications Security (CCS). 2021: 749-763.

[62] Sun H, Shen Y, Wang C, et al. Healer: Relation learning guided kernel fuzzing[C]//ACM Symposium on Operating Systems Principles (SOSP). 2021: 344-358.

[63] Sun H, Shen Y, Liu J, et al. KSG: Augmenting Kernel Fuzzing with System Call Specification Generation[C]//USENIX Annual Technical Conference (USENIX ATC 22). 2022: 351-366.

[64] Zhao B, Li Z, Qin S, et al. StateFuzz: System Call-Based State-Aware Linux Driver Fuzzing[C]//USENIX Security Symposium (USENIX Security). 2022: 3273-3289.

[65] Hao Y, Li G, Zou X, et al. SyzDescribe: Principled, Automated, Static Generation of Syscall Descriptions for Kernel Drivers[C]//IEEE Symposium on Security and Privacy (S&P). 2023.

[66] Bulekov A, Das B, Hajnoczi S, et al. No Grammar, No Problem: Towards Fuzzing the Linux Kernel without System-Call Descriptions.[C]//Network and Distributed System Security Symposium (NDSS). 2023.

[67] Fleischer M, Das D, Bose P, et al. ACTOR: Action-Guided Kernel Fuzzing[C]//USENIX Security Symposium (USENIX Security). 2023.

[68] Contributors S. syzkaller/syscall_descriptions_syntax.md at master[EB/OL]. GitHub. 2022 [2022-07-03]. https://github.com/google/syzkaller/blob/master/docs/syscall_descriptions_syntax.md.

[69] Michael M. Attack Landscape H1 2019: IoT, SMB traffic abound[EB/OL]. 2019. https://blog.f-secure.com/attack-landscape-h1-2019-iot-smb-traffic-abound/.

[70] Sattler J. Attack Landscape H2 2019: An unprecedented year for cyber attacks[EB/OL]. 2020. https://blog.f-secure.com/attack-landscape-h2-2019-an-unprecedented-year-cyber-attacks/.

[71] Eclypsium. Assessing Enterprise Firmware Security Risk in 2020[EB/OL]. 2020. https://eclypsium.com/2020-01-20-assessing-enterprise-firmware-security-risk/.

[72] Costin A, Zaddach J, Francillon A, et al. A large-scale analysis of the security of embedded firmwares [C]//USENIX Security Symposium (USENIX Security). 2014: 95-110.

[73] Vulnerability statistics of linux kernel[EB/OL]. https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html.

[74] Zhang Z, Zhang H, Qian Z, et al. An Investigation of the Android Kernel Patch Ecosystem[C]//USENIX Security Symposium (USENIX Security). 2021: 3649-3666.

[75] Jiang Z, Zhang Y, Xu J, et al. PDiff: Semantic-based Patch Presence Testing for Downstream Kernels [C]//ACM Conference on Computer and Communications Security (CCS). 2020: 1149-1163.

[76] Lawall J, Palinski D, Gnirke L, et al. Fast and Precise Retrieval of Forward and Back Porting Information for Linux Device Drivers[C]//2017 USENIX Annual Technical Conference (USENIX ATC). 2017: 15-26.

[77] Talebi S M S, Tavakoli H, Zhang H, et al. Charm: Facilitating dynamic analysis of device drivers of mobile systems[C]//USENIX Security Symposium (USENIX Security). 2018: 291-307.

[78] OpenWrt. OpenWrt Downloads[EB/OL]. 2020. https://downloads.openwrt.org/.

[79] OpenWrt. OpenWrt Archive[EB/OL]. 2020. https://archive.openwrt.org/.

[80] Testing Linux, one syscall at a time.[EB/OL]. https://linux-test-project.github.io/.

[81] Maier D, Radtke B, Harren B. Unicorefuzz: On the Viability of Emulation for Kernelspace Fuzzing [C]//USENIX Workshop on Offensive Technologies (WOOT). 2019.

[82] Hertz J, Newsham T. AFL/QEMU fuzzing with full-system emulation.[EB/OL]. 2016. https://github.com/nccgroup/TriforceAFL.

[83] Kuznetsov V, Kinder J, Bucur S, et al. Efficient State Merging in Symbolic Execution[C]//ACM Conference on Programming Language Design and Implementation (PLDI). 2012: 45-46.

[84] Cadar C, Dunbar D, Engler D R, et al. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs[C]//USENIX conference on Operating Systems Design and Implementation (OSDI). 2008: 209-224.

[85] Kroah-Hartman G. mtd: nand: orion: fix clk handling[EB/OL]. 2017. https://www.mail-archive.com/linux-kernel@vger.kernel.org/msg1403695.html.

[86] Yin H, Song D, Egele M, et al. Panorama: capturing system-wide information flow for malware detection and analysis[C]//ACM Conference on Computer and Communications Security (CCS). 2007: 116-127.

[87] Yan L K, Yin H. Droidscope: Seamlessly reconstructing the OS and dalvik semantic views for dynamic android malware analysis[C]//USENIX Security Symposium (USENIX Security). 2012: 569-584.

[88] Egele M, Kruegel C, Kirda E, et al. Dynamic spyware analysis[C]//2007 USENIX Annual Technical Conference (USENIX ATC). 2007: 233-246.

[89] Bayer U, Comparetti P M, Hlauschek C, et al. Scalable, behavior-based malware clustering[C]// Network and Distributed System Security Symposium (NDSS). 2009.

[90] Chipounov V, Kuznetsov V, Candea G. S2E: A Platform for in-Vivo Multi-Path Analysis of Software Systems[C]//International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). 2011: 265-278.

[91] Konovalov A. CVE-2017-1000112: Exploitable memory corruption due to UFO to non-UFO path switching[EB/OL]. 2017. https://www.openwall.com/lists/oss-security/2017/08-13/1.

[92] Contributors W. Virtual machine escape[EB/OL]. Wikipedia. 2021 [2021-10-06]. https://en.wikipedia.org/wiki/Virtual_machine_escape.

[93] Corporation T M. CVE Search Results matched “qemu”[EB/OL]. Mitre. 2021 [2021-10-06]. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=qemu.

[94] Cong K, Xie F, Lei L. Symbolic execution of virtual devices[C]//International Conference on Quality Software (ICQS). 2013: 1-10.

[95] Henderson A, Yin H, Jin G, et al. VDF: Targeted Evolutionary Fuzz Testing of Virtual Devices[C]//International Symposium on Research in Attacks, Intrusions, and Defenses (RAID). 2017: 3-25.

[96] Renatahodovan. renatahodovan/picire: Parallel Delta Debugging Framework[EB/OL]. GitHub. 2022 [2022-07-01]. https://github.com/renatahodovan/picire.

[97] Srivastava P, Payer M. GRAMATRON: Effective Grammar-Aware Fuzzing[C]//ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA). 2021: 244-256.

[98] Contributors Q. Fuzzing[EB/OL]. 2021. https://qemu.readthedocs.io/en/latest/devel/fuzzing.html.

[99] Team T C. Source-based Code Coverage[EB/OL]. 2021. https://clang.llvm.org/docs/SourceBasedCodeCoverage.html.

[100] Klees G, Ruef A, Cooper B, et al. Evaluating fuzz testing[C]//ACM Conference on Computer and Communications Security (CCS). 2018: 2123-2138.

[101] Lattner C, Lenharth A, Adve V. Making context-sensitive points-to analysis with heap cloning practical for the real world[C]//ACM Conference on Programming Language Design and Implementation (PLDI). 2007: 278-289.

[102] Microsoft. OpenHCI: Open Host Controller Interface Specification for USB[EB/OL]. Microsoft. 2022 [2022-11-29]. https://composter.com.ua/documents/OHCI_Specification_Rev.1.0a.pdf.

作者简历

教育经历

- 2018.09 - 2023.09，博士学位，计算机科学与技术学院，浙江大学
- 2014.09 -2018.06，学士学位，电子信息工程学院，北京理工大学

实习经历

- 2023.02 - 2023.08，访问博士学生，HexHive 实验室，洛桑联邦理工学院
- 2021.08 -2022.03，访问博士学生，HexHive 实验室，洛桑联邦理工学院

攻读博士学位期间主要的研究成果

- **Liu Q**, Toffalini F, Zhou Y, Payer M. VIDEZZO: Dependency-aware Virtual Device Fuzzing[C]// IEEE Symposium on Security and Privacy (S&P, CCF A). 2023.
- 周亚金, 刘强, 张岑, et al. 一种基于 QEMU 的嵌入式 Linux 内核动态分析平台: CN202110096520.7 [P]. (已进入实质审查，公开/公告日期：2021-06-04).
- **Liu Q**, Zhang C, Ma L, Jiang M, Zhou Y, Wu L, Shen W, Luo X, Liu Y, Ren K. FIRMGUIDE: Boosting the Capability of Rehosting Embedded Linux Kernels through Model-Guided Kernel Execution[C]//IEEE/ACM International Conference on Automated Software Engineering (ASE, CCF A). 2021.
- Ying J, Zhu T, **Liu Q**, Xiong C, Weng Z, Chen T, Fu L, Lv M, Wu H, Want T, Chen Y. TRAPCOG: An Anti-noise, Transferable, and Privacy-preserving Real-time Mobile User Authentication System with High Accuracy[J]. IEEE Transactions on Mobile Computing (TMC, CCF A). 2023.
- Jiang M, Ma L, Zhou Y, **Liu Q**, Zhang C, Wang Z, Luo X, Wu L, Ren K. ECMO: Peripheral transplantation to Rehost embedded Linux kernels[C]//ACM Conference on Computer and Communications Security (CCS, CCF A). 2021.
- Zhu T, Fu L, **Liu Q**, Lin Z, Chen Y, Chen T. One Cycle Attack: Fool Sensor-Based Per-

sonal Gait Authentication With Clustering[J]. IEEE Transactions on Information Forensics and Security (TIFS, CCF A), 2021.

- Zhu T, Weng Z, Song Q, Chen Y, **Liu Q**, Chen Y, Lv M, Chen T. ESPIALCOG: General, Efficient and Robust Mobile User Implicit Authentication in Noisy Environment[J]. IEEE Transactions on Mobile Computing (TMC, CCF A), 2020.